Process-based Requirement Derivation

Heather M. Conboy

May 4, 2010

Abstract. Computational agents are often utilized by human processes. So the requirements for the computational agents, such as a medical device, must take into account the overall process, such as a medical procedure, and the requirements that process must satisfy. But, the computational agent developers may not fully know the behavior of the overall process and even if they do it is challenging to reason about all potential behavior of the process since it is usually large and complex. Therefore, it is difficult to gather all of the requirements of a computational agent that must be satisfied for the process to satisfy its requirements. Thus, we investigated an automated requirement derivation approach that inputs a process model and a process requirement, exhaustively considers all potential behavior of the process, and outputs a derived requirement of the computational agent that is sufficient to prevent any violations of the process requirement. We extended an assumption derivation algorithm, developed to support assume-guarantee reasoning techniques, to automate the derivation of the computational agent requirements. The approach iteratively derives the requirements by employing a learning algorithm in combination with a finite state verification tool. We implemented the automated requirement derivation approach and evaluated it by applying it to two case studies in the medical domain. This preliminary evaluation showed that the approach could be successfully applied to abstract models of portions of real processes and requirements of those processes that involve safety and security. For computational agent developers, such as software engineers and security analysts, the derived requirements appeared useful and understandable.

1 Introduction

Human-intensive processes (HIPs) manage important tasks such as banking, elections, and health care. HIPs are often large and complex, involving features such as concurrency, synchronization, and exceptional control flow, which makes them error-prone. A HIP's requirements describe how that HIP should behave, addressing critical aspects such as safety, security, and privacy. When a HIP satisfies its requirements, we gain assurance that the HIP performs its task correctly. If a HIP fails to satisfy its requirements, it could cause significant harm.

A HIP can be viewed as being composed of a process coordinator component, which is responsible for the ordering of and communication among subtasks, and agent components, which perform those subtasks. The agent components can be human agent components or computational agent components (e.g. hardware devices, software applications). For brevity, the HIP component names will be hereafter shortened to process coordinator and agents. A HIP satisfies its requirements only if the composition of the process coordinator's behavior and the agents' behaviors satisfies that HIP's requirements.

A new computational agent (e.g., bank security module, voting machine, implantable medical device) is developed to satisfy certain computational agent requirements. But often those computational agent requirements are determined in isolation, without taking into account the process coordinator or the HIP's safety, security, and privacy requirements. If the computational agent requirements are not adequate, then when the released computational agent is composed with the process coordinator to form a HIP, some or all of the HIP's safety, security, and privacy requirements may be violated. But even when the process coordinator's behavior and the HIP's requirements are known, determining computational agent requirements that are adequate is challenging. Since the coordination is often complex, it is easy for some crucial computational agent requirement to be missed or inaccurately specified, but this may not be discovered until late in the computational agent development cycle, perhaps only after the released computational agent is deployed in a HIP. The usual computational agent development cycle is design, implement, test, and release. The further along the development cycle any requirement violations are discovered, the more expensive it is to fix them. Therefore, we also want the computational agent requirements to become adequate as early in the development cycle as possible to reduce the development costs.

To illustrate, consider a hospital care HIP that manages an in-patient surgery. The hospital care HIP is composed of a process coordinator responsible for the ordering and communication among the in-patient surgery subtasks (e.g. the operation itself, post-operative care, administration of fluids and medications) and a computational agent such as a "smart" infusion pump, which is responsible for intravenously administering fluids and medications (i.e. drugs). For brevity, a "smart" infusion pump will hereafter be shortened to a pump. For such pumps, each primary care area is associated with a drug library that enumerates the drugs in use in that area. Each drug in use is associated with the usual dosing parameters such as concentrations, dosing units, and dosing limits. Within the operating room, the pump is commonly configured with a drug library that allows a wider range of dosing limits. But within the intensive care unit, the pump is usually configured with a drug library that allows a more restrictive range of dosing limits. One possible HIP requirement is that a patient is never administered a drug over/under dose (i.e. a dose that exceeds the allowable dosing limits in a particular primary care area). So if the health care practitioners move a pump from the operating room to the intensive care unit without reconfiguring the pump (or vice versa), then the HIP requirement regarding never giving an over/under dose could be violated. If the pump developers do not consider a HIP where a pump is moved from one primary care area to a different primary care area, then they could miss requirements of the pump. In this case, the pump developers could miss the requirement of the pump that captures that the pump must be reconfigured when it is moved into a primary care area with a more restrictive range of dosing limits.

Computational agent developers, both software engineers and security/privacy analysts, have generally found that checking for safety/security/privacy violations "is complicated by the fact that they often exist in hard-to-reach states or crop up in unusual circumstances" [11]. In particular, the computational agent developers could miss possible safety/security/privacy violations by not considering the hard-to-reach states or the unusual circumstances. Thus, an automated requirement derivation approach that inputs a process coordinator and a HIP safety, security, or privacy requirement and exhaustively checks for safety/security/privacy violations, including the hard-to-reach states or the unusual circumstances, would be of assistance. Additionally, if such an automated requirement derivation approach discovered a safety/security/privacy violation, then such an automated approach could output a derived requirement of the computational agent that would be sufficient to prevent the violation. That is, if the computational agent were modified to satisfy this derived requirement, the HIP composed of the given coordinator and the modified computational agent would not violate the HIP requirement. Such an automated requirement derivation approach would be an automated assistant that checks for safety/security/privacy violations and outputs if any violations are discovered and if so also outputs a derived requirement of the computational agent. But, the computational agent developers would still be responsible for specifying the HIP requirements and deciding how to modify the computational agent to satisfy the derived requirement.

To illustrate consider the case where the process coordinator allows the pump to be moved from the operating room to the intensive care unit and the requirement that the pump prevent administration of a drug over/under dose. Such an approach might derive the requirement that, after the pump is moved to a new primary care area, it must be reset before any medications can be infused. In general, if such an automated requirement derivation approach outputs that the HIP satisfies the HIP requirement, then the computational agent's behavior can remain as is. Otherwise if the HIP violates the HIP requirement then the computational agent's behavior can be modified based on the derived requirement of the computational agent.

For this project, we investigated such an automated requirement derivation approach that inputs an existing process coordinator and a HIP safety, security, or privacy requirement and then outputs whether the HIP satisfies the HIP requirement and additionally a derived requirement of the computational agent. We hypothesized that assumption learning algorithms developed to support assume-guarantee reasoning techniques could be extended to automate the derivation of the computational agent requirements. To the best of our knowledge, such an automated requirement derivation approach has not been evaluated for HIPs and their requirements. Additionally, we know of no automated requirement derivation approach that applies to requirements involving safety, security, and privacy. Existing requirement derivation approaches either consider only safety or only security and privacy. Further, some of the existing requirement derivation approaches are only semi-automated since they expect the user to incrementally provide more input as the derivation proceeds.

Our contributions are the automated requirement derivation approach, a requirement derivation toolset that implements that approach, and a preliminary evaluation of the approach by applying the toolset to two case studies. For the case studies, the process coordinator is written in a process modeling language, specifically the Little-JIL process definition language [36]. The HIP requirements are specified in a requirement specification language, specifically finite state automata (FSAs). The derived computational agent requirements are also represented as FSAs. For the preliminary evaluation, we primarily investigated and evaluated whether or not the automated requirement derivation approach:

1. can be applied to "real" HIPs (i.e. what is the performance in terms of space and time)
2. can be applied to safety, security, and privacy requirements and, if possible, whether it is useful to do so
3. can derive requirements of the computational agent that are readily understandable and useful to computational agent developers (i.e. fill in missed or correct inaccurate requirements of the computational agent)

Overall, the automated requirement derivation approach shows promise with regards to the above evaluation criteria. But, further improvement and evaluation is needed.

Section 2 provides background on the assumption learning algorithms, Section 3 describes the proposed automated requirement derivation approach, Section 4 describes the requirement derivation toolset that was developed to investigate the automated requirement derivation approach while Section 5 describes the evaluation using that toolset. Section 6 discusses related work and Section 7 concludes and briefly discusses future work.

2 Background

Finite state verification (FSV) approaches were developed to verify whether or not a hardware and/or software system satisfies a requirement. Specifically, FSV approaches verify algorithmically that all potential executions of a finite-state model of a system satisfies a requirement. If so, then the verification result is that the requirement is satisfied. If not, then the verification result is that the requirement is violated and a counter example execution that demonstrates how the finite-state model of the system violates the requirement is provided. The FSV approaches, however, suffer from the state explosion problem where the size of the finite-state model or the cost of the verification algorithm may grow exponentially with the size of the system. There exist various FSV tools that implement the FSV approaches. Often the FSV tools incorporate many optimizations to combat the state explosion problem.

Instead of verifying an entire system at once, compositional verification approaches tackle the state explosion problem with a divide and conquer strategy. Intuitively, such approaches take advantage of a system being composed of system components and verify each system component separately. An individual system component, however, often satisfies a requirement only in certain environments. Assume-guarantee reasoning (AGR) techniques have been developed to utilize a provided *assumption* about the environments in which a system component is used. As described in [14], the key idea is that if the given system component is part of a system that satisfies the provided assumption, then the system also guarantees the requirement. Essentially, the provided assumption restricts the environment's behavior so that the system component's behavior satisfies the requirement. Thus, there exist compositional verification approaches that verify the overall system by employing the AGR techniques to verify each system component separately.

For AGR techniques, the simplest case is when the system is composed of only two components: a system component and a particular environment of that system component. The simplest AGR proof rule consists of two steps. The first step verifies that the system component satisfies the requirement when used in any environment that satisfies the provided assumption. The second step verifies that the particular environment satisfies the provided assumption. If both steps are true, then the entire system composed of the system component and the particular environment satisfies the requirement. But, the assumptions are difficult to provide. As described in [14], AGR techniques must be provided with assumptions "that are strong enough to eliminate false violations, but that also reflect the remaining system appropriately." If an assumption is too weak (not restrictive enough) then the environment's actual behavior is overapproximated so the requirement is violated by environment behavior that is infeasible. Alternatively, if that assumption is too strong (overly restrictive) then the requirement is satisfied but more of the environment's actual behavior could be allowed without the requirement being violated.

Since the assumptions are difficult to provide, assumption learning algorithms (e.g. [3, 9, 14]) were developed that perform the compositional verification by employing the AGR techniques while *learning* the assumption. At a high-level on each iteration, an assumption learning algorithm considers a learned assumption A_i. If the algorithm can determine the verification result with A_i, then it stops iterating. If not, the algorithm continues learning the assumption by essentially first making the assumption strong enough so that the requirement is satisfied and secondly making the assumption weak enough so that the environment satisfies the assumption. In more detail on each iteration, an assume learning algorithm performs two phases. In phase 1, the algorithm checks AGR step 1 that verifies whether or not the system component satisfies the requirement when restricted by assumption A_i. If the system component violates the requirement, then the assumption must become stronger to satisfy the requirement. The provided counter example is used to modify the assumption, specifically to disallow the counter example behavior of the environment. The algorithm continues iterating since the assumption may now be too strong so needs to be become weaker. In phase 2, if the system component satisfies the requirement when restricted by assumption A_i, then the algorithm checks AGR step 2 that verifies whether or not the environment satisfies assumption A_i. If the environment satisfies the assumption, then the compositional verification result is that the overall system satisfies the requirement. Otherwise if the environment violates the assumption, then the algorithm must check whether or not the provided counter example corresponds to an actual counter example in the overall system. For the actual counter example execution check, if the provided counter example satisfies the requirement, then it is not an actual counter example so the assumption must become weaker. The provided counter example is used to modify the assumption, specifically to allow the counter example behavior of the environment. The algorithm continues iterating since the assumption may now be too weak so needs to become stronger. Otherwise if the provided counter example violated the requirement, then it is an actual counter example so the compositional verification result is that the overall system violates the requirement.

At a lower-level, the assumption learning algorithms employ the L* algorithm in combination with an FSV tool to automatically learn the assumption represented as an automaton. The L* algorithm, by Angluin [5] and improved by Rivest and Schapire [31], learns an unknown regular language U over an alphabet Σ and returns a minimal deterministic FSA A such that $L(A)$ is equivalent to U. For the AGR techniques, U should correspond to an assumption that is strong enough so that the requirement is satisfied (i.e. AGR step 1 is true). In addition, U should ideally be weak enough so that the environment satisfies U (i.e. AGR step 2 is also true). The L* algorithm learns by interacting with a "minimally adequate teacher" that is essentially an oracle that is capable of answering queries about U. The teacher must be able to answer two types of queries. Query type 1 is a membership query that inputs a string s from Σ^* and checks whether or not s is in U. When s is in U, it outputs **true**. Otherwise when s is not in U, it outputs **false**. Query type 2 is an equivalence query that inputs a conjectured FSA A_i and checks whether or not $L(A_i)$ is equivalent to U. When $L(A_i)$ is equivalent to U, it outputs true. Otherwise when $L(A_i)$ is not equivalent to U, it outputs false and a counter-example string from the set difference of $L(A_i)$ and U. We describe how the automated requirement derivation approach extends an assumption learning algorithm in the next section.

3 Automated Requirement Derivation Approach

At a high-level, the automated requirement derivation approach inputs a HIP and a HIP requirement and outputs the derived requirement imposed on the computational agent. We create a HIP composed of the existing process coordinator and a newly created pessimistic computational agent that captures all potential behavior of the computational agent. In particular, the pessimistic computational agent is allowed to perform its subtasks in any order or even not at all. We want the pessimistic computational agent to capture all potential behaviors of the computational agent so that the requirement derivation can learn what restrictions must be placed on these behaviors of the computational agent so that the HIP satisfies its HIP requirement.

More formally, HIP P is composed of a process coordinator C and a set of agents $D_1, \ldots, D_n$. The composition will be represented as $P = C(D_1, \ldots, D_n)$. To simplify the problem, we made two assumptions. The first assumption is that all human agents perform their subtasks correctly. The second assumption is that there is a single pessimistic computational agent D. So this composition will now be represented as $P = C(D)$. The proposed automated requirement derivation approach inputs a HIP P, composed of an existing process coordinator C and a pessimistic computational agent D, and a HIP requirement R_P. It extends one of the assumption derivation algorithms so that the output is a derived computational agent requirement that restricts the behavior of D so that $P = C(D)$ satisfies R_P.

None of the assumption learning algorithms can be applied as is because the pessimistic computational agent allows all potential behaviors which usually include violating behaviors. Thus, the HIP requirement is almost always violated. Since the assumption learning algorithms stop immediately when the verification detects a violation of a HIP requirement, such algorithms often stop too soon for the learning to take place. This means that the learned assumption is strong enough to ensure that the HIP requirement is satisfied but is usually too strong to be a useful computational agent requirement. Therefore, we plan to extend one of the assumption learning algorithms to make the learned assumption strong enough to ensure that the HIP requirement is satisfied but weaker so that it can be used as a computational agent requirement.

At a high-level, the extension is to not stop immediately when a violation of a HIP requirement is detected but instead to continue learning. Thus, the requirement imposed on the computational agent now essentially allows all potential behaviors of the computational agent that satisfy the HIP requirement and disallows the remaining potential behaviors that violate the HIP requirement. Additionally, the user is permitted to provide initial requirements of the computational agent that restrict the computational agent to more reasonable behavior (e.g. the computational agent must be turned on before performing any other subtasks). The computational agent developers would then be provided with the derived requirements of the computational agent and any initial requirements of the computational agent. We describe the original assumption learning algorithms first and then the extended assumption learning algorithm second.

3.1 Original Assumption Learning Algorithms

The assumption learning algorithms learn an assumption represented as an automaton. They employ the L* algorithm where the teacher answers the membership and equivalence queries by employing an FSV tool. The L* algorithm incrementally learns an unknown language U with alphabet Σ and outputs an automaton C that accepts that language (i.e. $L(C) = U$). At a high-level, L* stores a table that records whether or not strings from $\Sigma*$ are contained in U. First, it initializes the table using membership queries answered by the teacher. On each iteration i, L* updates the table using membership queries answered by the teacher. Next, it uses the table to conjecture an automaton C_i. L* checks whether $L(C_i)$ is equal to U by using an equivalence query answered by the teacher. If so, then L* stops iterating and outputs C_i. Otherwise L* continues iterating after it updates the table based on the counter example returned by the teacher using membership queries answered by the teacher.

L* algorithm Specifically, L* stores its information in an observation table (S,E,T) where conceptually S is a set of prefixes represented as strings over $\Sigma*$, E is a set of suffixes represented as a set of strings over $\Sigma*$, and T is which strings over $\Sigma*$ are contained in U represented as a function from $(S \cup S \cdot \Sigma) \cdot E$ to $\{true, false\}$. For L*, the function Learn(Σ) pseudo code is: //TODO: Fix indentation below

```
1: $S \leftarrow \{\lambda\}; E \leftarrow \{\lambda\};$
2: $Update(T, S, E);$
3: **while** (true) **do**
4: **while** $((s_{new} \leftarrow CheckClosed(S, E, T)) \neq null)$ **do**
5: $S \leftarrow S \cup \{s_{new}\};$ // New prefix
6: **foreach** $(a \in \Sigma), (e \in E)$ **do**
```

```
7: $T[s_{new} \cdot a, e] \leftarrow MembershipQuery(s_{new} \cdot a \cdot e)$;
8: $C_i \leftarrow MakeConjecture(S, E, T)$;
9: **if** $((cex \leftarrow EquivalenceQuery(C_i)) = null)$
10: **then return** $C_i$;
11: **else**
12: $e_{new} \leftarrow FindSuffix(cex); E \leftarrow E \cup \{e_{new}\}$; // New suffix
13: $Update(T, S, \{e_{new}\})$;
```

L* is responsible for the functions Update, CheckClosed, MakeConjecture, and FindSuffix. The teacher is responsible for the functions MembershipQuery and EquivalenceQuery. We describe the L* functions are first and the teacher functions second.

To begin, L* initializes the observation table. S and E are set to $\{\lambda\}$ and T sets whether the zero and one length strings over $\Sigma*$ are in U (lines 1, 2). The function Update(T,S,E) pseudo code is:

```
14: **foreach** $(s \in S), (a \in \Sigma), (e \in E)$ **do**
15: $T[s, e] \leftarrow MembershipQuery(s \cdot e)$;
16: $T[s \cdot a, e] \leftarrow MembershipQuery(s \cdot a \cdot e)$;
```

Next, L* iterates (line 3). On each iteration, L* repeatedly updates the observation table (lines 4,5,6,7) until the table is closed. The function CheckClosed(S,E,T) checks whether the table meets the condition for all s in S and a in Σ there exists an s' in S such that $T[s \cdot a, e]$ equals $T[s', e]$ for all e in E. If so, then it returns true. Otherwise, it returns false and the string $s \cdot a$ that did not meet the condition. In more detail, L* first checks whether or not the table is closed (line 4). If the table is not closed, then L* updates the table using the new prefix $s \cdot a$ (lines 5,6,7). If the table is closed, then L* makes a conjectured automaton C_i based on the table (line 8). The function MakeConjecture(S,E,T) creates an automaton with alphabet Σ where the states are created based on S and the transitions are created based on T. Next, L* checks whether or not $L(C_i)$ is equal to U (line 9). If so, then it outputs C_i. Otherwise, L* finds a new suffix e_{new} based on the counter example string cex and adds that new suffix to E (line 12). The function FindSuffix(cex) analyzes the counter example cex to "find a suffix e that witnesses a difference between $L(C_i)$ and U; e must be such that adding it to E will cause the next conjectured automaton to reflect that difference" [14]. The function FindSuffix(cex) described in more detail in [31]. Lastly, L* continues to the next iteration after updating the table using that new suffix (line 13).

The L* algorithm provides two guarantees. The first guarantee is that it will terminate and the second guarantee is that it will return a minimal deterministic FSA for the unknown language U. The number of membership queries made is $O(|\Sigma|n^2 + n \log m)$ where n is the number of states in the returned FSA and m is the length of the longest counter-example output by any equivalence query. The number of equivalence queries is at most n - 1.

Teacher that employs an FSV tool For the assumption learning algorithms, the teacher inputs a software system and a requirement. It provides the functions MembershipQuery and EquivalenceQuery. From Section 2, the software system is decomposed into a software component and its environment. The teacher employs an FSV tool. Recall that an FSV tool checks that all potential executions of a FSV model of a software system satisfy a requirement. If so, then the FSV tool reports that the requirement is satisfied. If not, then the FSV tool reports that the requirement is violated and provides a counter example execution that demonstrates the violation. The assumption learning algorithms stop iterating as soon an actual counter example execution is discovered.

For the function MembershipQuery(s), the teacher basically checks whether or not string s is contained in the language of the requirement. If so, then it outputs **true**. If not, then it outputs **false**.

For the function EquivalenceQuery(C_i), the teacher employs the FSV tool to first check if the assumption is strong enough (query type 2 a) and then to check if the assumption is weak enough (query type 2b). For query type 2 a, the teacher employs the FSV tool to check whether or not the system component satisfies the requirement when restricted by the conjectured FSA A_i. If the FSV tool reports that the requirement is violated then the teacher outputs **false** and a counter example string from $L(A_i)\backslash U$ where the counter example string is created from the counter example execution provided by the FSV tool. If the FSV tool reports that the requirement is satisfied, then the teacher continues on to query type 2 b.

For query type 2 b, the teacher employs the FSV tool to check whether or not the environment satisfies the assumption. If the FSV tool reports that the environment satisfies the assumption, then the teacher outputs **true**. If the environment violates the assumption, then the FSV tool is employed to check if the counter example execution provided by the FSV tool is an actual counter example.

To check the counter example, the teacher basically checks whether or not the provided counter example execution is contained in the language of the requirement. If so, then the provided counter example is an actual counter example so the teacher outputs **true**. If not, then the provided counter example is not an actual counter example so the teacher outputs **false** and a counter example string from $U \setminus L(A_i)$ where the counter example string is created from the provided counter example execution.

Cobleigh et al [13] performed the most extensive evaluation of an assumption learning algorithm by applying it to concurrent software systems and their requirements. The evaluation showed that the assumption learning performance varied greatly (e.g. time ranged from 20 seconds to being stopped after a couple of hours) and also the complexity of the learned assumptions varied widely. For sequential Java programs, Alur et al [4] and later Beyer et al [8] investigated an automated requirement derivation approach that inputs an existing library class (e.g. File) that has already been released and a safety requirement (e.g. IOException is not thrown) and then outputs the requirement imposed on any clients of that library. The client requirement derivation approaches also extended the assumption learning algorithms to automate the client requirement derivation. For the client requirement derivation approaches, the evaluation was not as extensive but also showed that the requirement derivation performance and the complexity of the derived requirements could vary widely.

3.2 Extended Assumption Learning Algorithm

The proposed automated requirement derivation approach extends an assumption learning algorithm similarly to what was done in [4,8]. At a lower-level, the requirement derivation algorithm is the same as the assumption learning algorithm with regard to the L* functions Learn, CheckClosed, Update, MakeConjecture, and FindSuffix. It is also the same with regards to the teacher function MembershipQuery. But it extends the teacher function EquivalenceQuery. At a high-level, the extension is to continue to iterate even after one or more actual counter examples are discovered in an effort to weaken the assumption and make it a more useful derived requirement.

Teacher that employs an FSV tool The teacher now records the compositional verification result as *not yet determined*, *satisfied*, or *violated*. Initially, it records the compositional verification result as *not yet determined*. For query type 1 a membership query, it remains unchanged. For query type 2 an equivalence query, it is changed, specifically query type 2 (a) remains unchanged while query type 2 (b) was extended as follows.

If the system component satisfies the requirement when restricted by assumption A_i, then the teacher checks AGR step 2 by employing the FSV tool to verify whether or not the environment satisfies assumption A_i. If the environment satisfies the assumption, then the overall system satisfies the requirement so the teacher records the compositional verification result as *satisfied* and outputs **true**. Otherwise the environment violated the assumption, so the teacher must check whether or not the provided counter example corresponds to an actual counter example in the overall system.

For the actual counter example execution check, if the provided counter example satisfies the requirement, then the assumption must become weaker. The teacher outputs **false** and also the provided counter example. The provided counter example will be used to modify the assumption, specifically to allow the counter example behavior of the environment. The L* algorithm continues iterating since the assumption may now be too weak so needs to become stronger. Otherwise if the provided counter example violates the requirement, then the overall system violates the requirement so the teacher records the compositional verification result as *violated*.

The L* algorithm will stop iterating if no more counter examples are provided. Specifically to weaken the assumption, we want a counter example execution where the assumption is violated but the requirement is satisfied. Therefore, the teacher employs the FSV tool to verify whether or not the environment satisfies the assumption *when restricted by the requirement*. If so, the teacher outputs **true** and thus provides no more counter examples. Otherwise, the teacher outputs **false** and also provides a counter example that is used to weaken the assumption as described in the previous paragraph. In the next section, we describe a requirement derivation toolset that implements automated requirement derivation approach described in this section.

4 Requirement Derivation Toolset

To evaluate the automated requirement derivation approach, the requirement derivation toolset was applied to two case studies as described in Section 5.1. For the requirement derivation toolset, we needed process models that are expressive enough to capture the HIPs and precise enough to be formally analyzed by tools. Additionally, we needed requirements specifications that are expressive enough to capture the HIP and computational agent requirements and also precise enough to be formally analyzed by tools and readily understood by developers. For the requirement deriver tool, we needed an assumption learning algorithm to extend to automate the requirement derivation. For that assumption learning algorithm, we also needed an FSV tool to answer the teacher's queries.

The HIP is defined in a process modeling language. Process modeling languages commonly have language constructs that support nominal control flow and data flow but not as typically have language constructs that support exceptional control flow and synchronization. The process modeling languages sometimes are precise enough to be formally analyzed. We used the Little-JIL process definition language [36], which is both expressive and precise. It has successfully been used to define an election HIP and a health care HIP (e.g., [10, 33]). There exists a translation from a Little-JIL process model to a finite-state model that is based on control flow graphs (CFGs). The translation includes several optimizations. For the requirement derivation toolset, the translation to a finite-state model was extended to be able to define the HIPs.

The HIP requirements and computational agent requirements (including any initial and the derived) are described in a requirement specification language. Requirement specification languages are often either a natural language, like English, or a mathematical formalism such as temporal logic or finite state automata (FSAs). We used FSAs, which are both expressive and precise. Specifically, we used deterministic FSAs that are total. Therefore, for every state s in the FSA for every event e in the alphabet of the FSA there exists one and only one transition where the source state is s and the event is e.

For the requirement deriver tool, we used the assumption derivation algorithm from [14] so that the learned assumptions are represented as FSAs. For the automated requirement derivation approach, the assumption learning algorithm was extended as described in Section 3.2 so that the learned assumption ensures that the HIP requirement is satisfied while placing fewer restrictions on the computational agent. For the FSV tool, we used FLAVERS [16] for Little-JIL that verifies whether or not a process model defined in Little-JIL satisfies a requirement represented as an FSA. It incorporates various optimizations. In what follows, FLAVERS for Little-JIL will be called FLAVERS/Little-JIL. It has successfully been applied to both an election HIP and a health care HIP to verify that they satisfy requirements about their safety (e.g., [10, 33]). For the requirement derivation toolset, FLAVERS/Little-JIL also was extended to be able to capture the HIPs semantics more accurately.

Section 4.1 provides a brief overview of Little-JIL, Section 4.2 provides a brief overview of FLAVERS/Little-JIL, and Section 4.3 describes each requirement derivation tool in more detail.

4.1 Little-JIL Overview

A Little-JIL process definition precisely captures a process that manages a particular task where the process is composed of agents that are responsible for performing the subtasks and the coordinator that is responsible for the ordering among the subtasks and any communication among the agents. From [10], "a Little-JIL process definition has three components, an artifact collection, a resource repository, and a coordination specification." In the following description, we focus on the aspects of Little-JIL that are used by the case studies described in Section 5.1. The resource repository defines which agents perform the subtasks. The artifact collection defines what artifacts are consumed and/or produced by the subtasks. A coordination specification precisely defines how the agents perform the subtasks that consume and/or produce the artifacts. It has a visual representation. The Little-JIL language reference [36] contains a more detailed discussion.

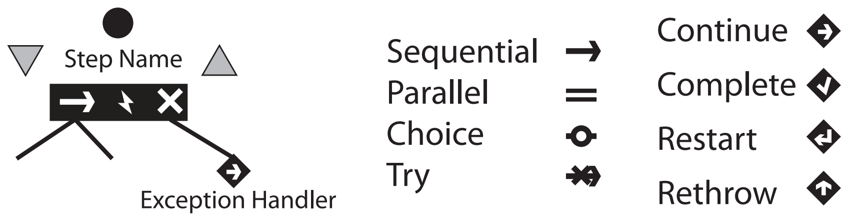

Figure 1: Little-JIL step bar with sequencing badges and continuation badges

The Little-JIL coordination specifications support high-level features such as abstraction, parallelism, iteration, and exceptional control flow. Each coordination specification consists of a hierarchical decomposition of steps where each step represents a subtask to be performed by a particular agent. A step is declared once but may be referenced multiple times. Figure 1 shows how a step declaration is depicted within a coordination specification as a step bar with the step name above the step bar. The resources/artifacts are depicted by the interface badge (i.e. the circle above the step bar). A non-leaf step declares children steps depicted by the outgoing arcs from that step. Additionally, a non-leaf step must specify the ordering among the children depicted by the sequencing badge (i.e. the symbol on the left hand side within the step bar). In Figure 1, the sequencing badges are enumerated in the middle. For example, parallelism is depicted by an equal sign on the left hand side within the step bar. A leaf step declares no children. The leaf steps are executed by the agents. “A step is reasonably thought of as a procedure” [33].

As mentioned in the introduction, the agents can be human agents or computational agents. We focus on the computational agents. The artifacts used in the case studies are channels, in/out parameters, and exceptions. Two or more agents may communicate over a channel where a channel is essentially a bounded buffer that stores data. The agents may communicate by either writing a datum to a channel or taking a datum from a channel. A step may store data by declaring in/out parameters. The parameters may be copied between a step and its children (i.e. defined and used). “As steps can be thought of as procedures, this artifact passing is essentially a parameter passing mechanism” [33]. Additionally, a step may throw exceptions. The exceptions are then propagated through the step hierarchy and another step may catch one or more of those exceptions, handle them, and specify where the control proceeds designated by the continuation badge. In Figure 1, the continuation badges are enumerated on the right hand side.

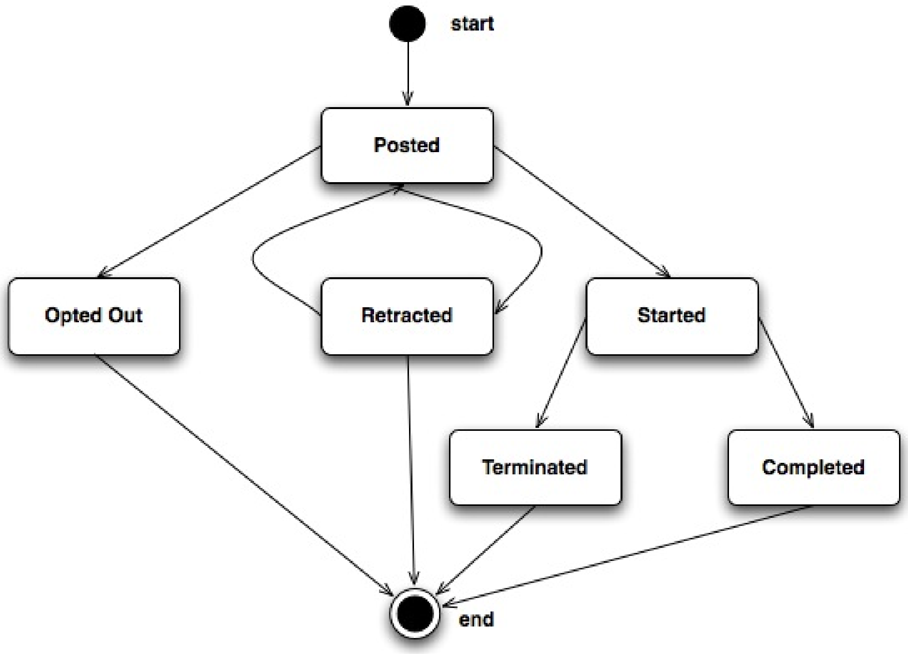

Figure 2: Little-JIL step execution life-cycle represented as a UML state diagram

Within the coordination specification that defines a process, a single step must be defined as the entry point to that process depicted by a northeast arrow to the right of the step name. When a process is executed, the entry point step is executed first. A step is in one of the following execution states: POSTED, STARTED, COMPLETED, TERMINATED, RETRACTED, or OPTEDOUT. From [36], Figure 2 shows the step execution life-cycle represented as a UML (Unified Modeling Language) state diagram. For the nominal control flow, the typical sequence of states is POSTED, STARTED, COMPLETED while for the exceptional control flow, the usual sequence of states is POSTED, STARTED, TERMINATED. The RETRACTED state is primarily relevant for choice steps while the OPTEDOUT state is primarily relevant for steps that an agent elected to opt out of performing.

4.2 FLAVERS/Little-JIL Overview

Given a Little-JIL process and a user-defined property, FLAVERS/Little-JIL uses a data flow algorithm to verify that the process satisfies the property. FLAVERS/Little-JIL is event-based. Each event represents a process sub-task such as a step starting, a step throwing an exception, a step defining a parameter, or a step communicating over a channel. FLAVERS is described in full detail in [16].

The property captures the event sequences that should (or should not) occur on any execution of the process. FLAVERS/Little-JIL internally represents the property as an FSA where the transitions are annotated with the events. The Little-JIL process is represented as a trace flow graph (TFG). Essentially, the TFG consists of a collection of Control Flow Graphs (CFGs) where each CFG represents an in-lined thread and the nodes are labeled with the events. Extra nodes and/or edges are also added to capture the thread synchronizations and inter-leavings. The paths through the TFG are represented as sequences of the nodes and therefore correspond to event sequences. These paths represent the potential executions of the process. Thus, the data flow algorithm verifies that all paths through the TFG satisfy the property.

The TFG is a conservative but imprecise model of the process. In the TFG, there exist feasible paths that do correspond to actual executions of the process. But in addition because the TFG is imprecise, there may exist infeasible paths that do not correspond to actual executions of the process. If the TFG model of the process is too imprecise to verify the property, then feasibility constraints may be employed to model additional information about the process. Each constraint captures the event sequences that must (or must not) occur on any potential execution of the process. FLAVERS/Little-JIL internally represents each constraint as an FSA where the transitions are annotated with the events. FLAVERS/Little-JIL provides support to automatically build thread related constraints that capture a particular thread's execution status [28] or that thread's program counter [16]. Additionally, it provides support to automatically build variable constraints that capture a particular variable's value where that variable may be defined (e.g. event "pumpReqChn=0") and used (e.g. event "pumpReqChn!=1"). Thus, the data flow algorithm verifies that all paths through the TFG that adhere to the constraints satisfy the property.

The state propagation algorithm performs the data flow analysis. The TFG supports single-entry/single-exit semantics where the single-entry point is represented by a unique initial node and the single-exit point is represented by a unique final node. During the state propagation algorithm, each TFG node is associated with a set of tuples where a tuple has a position for the property's state and for each constraint's state. The TFG's initial node is associated with the initial tuple that has the property at its start state and each constraint at its start state. The data flow algorithm propagates the tuples among the nodes until a fixed point is reached. The TFG's final node is associated with the final set of tuples. To determine the verification result, the final tuples are considered. A final tuple adheres to the constraints when each constraint is at an accepting state. If all final tuples that adhere to the constraints also have the property at an accepting state, then the process satisfies the property. The state propagation algorithm has worst case complexity that is $O(N^2 \cdot P \cdot C_1 \cdot \ldots \cdot C_m)$ where N is the number of nodes in the TFG, P is the number of states in the property automaton, m is the number of constraints, and C_i is the number of states in constraint automaton i.

4.3 Requirement Derivation Tools

Figure 3 shows the proposed architecture of the requirement derivation toolset where each tool is represented by a rectangle and the data flow among the tools is represented by the arrows. For now,

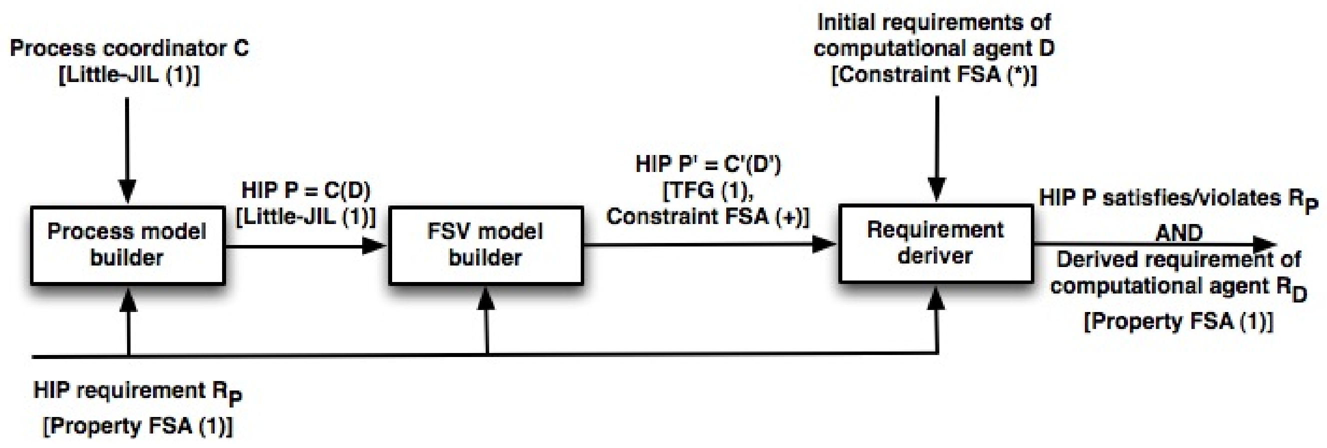

Figure 3: Architecture of the Requirement Derivation Toolset

we manually built the process models. The implemented requirement derivation toolset automated the FSV model building and the requirement derivation. First, the process model builder tool inputs an existing HIP process coordinator model C written in Little-JIL and a safety, security, or privacy HIP requirement R_P represented as a property FSA and outputs a newly created HIP process model P, composed of the existing HIP process coordinator model C and a newly created pessimistic computational agent model D, written in Little-JIL. Second, the FSV model builder inputs the HIP model process model P and the HIP requirement R_P and outputs a HIP FSV model P' represented as a TFG and also a set of one or more constraint FSAs. Lastly, the requirement deriver implements the extended assumption learning algorithm described in Section 3.2. It inputs the HIP FSV model P', the HIP requirement R_P, and optionally any initial requirements of the computational agent's behavior represented as a set of constraint FSAs. It outputs whether or not the HIP satisfies the safety, security, or privacy HIP requirement and additionally a derived requirement of the computational agent R_D represented as a property FSA.

For [10, 33], the FLAVERS/Little-JIL front-end input a HIP that contained only the process coordinator and output the corresponding TFG. If within the process coordinator, there existed leaf steps to be executed by a computational agent, then they were treated as remote procedure calls (RPCs) to that computational agent. But, the computational agent was not explicitly represented in the TFG and neither were the RPCs. For the automated requirement derivation approach, we want to allow the user to provide any initial requirements on the computational agent's behavior so the computational agent must be explicitly represented in the TFG and so must the RPCs. Therefore, a HIP is composed of a process coordinator and a computational agent so that the corresponding TFG explicitly represents the computational agent's behavior. Additionally, a HIP must encode the RPCs for the corresponding TFG to represent them.

We elected to encode the RPCs with channels. For a particular computational agent, its RPCs are encoded with two channels, a request channel and a response channel. First, a process coordinator sends a request over the request channel where that request encodes the requested procedure name and any in parameters. Second, the computational agent receives a request from the request channel and decodes that request. Next, it executes the named procedure on the given in parameters. Third, the computational agent sends a response to the response channel where the response encodes any out parameters or exceptions thrown. Fourth, the process coordinator receives a response from the response channel and decodes that response. Conceptually, the derived requirement of the computational agent partitions the sequences of RPCs to that computational agent into the sequences that satisfy the HIP requirement and the remaining sequences that violate the HIP requirement. In the following sections, we describe each requirement derivation tool in more detail.

4.3.1 Process model builder tool

The following sketches out what a process model builder tool would do. A new HIP process model is built by composing an existing process coordinator model and a newly created pessimistic computational agent model. For now, we manually built the HIP process models. In the future, we plan to implement a process model builder tool that automatically builds the HIP process models. First, we describe the new HIP model. Second, we describe the process coordinator model. Lastly, we describe the new pessimistic computational agent model.

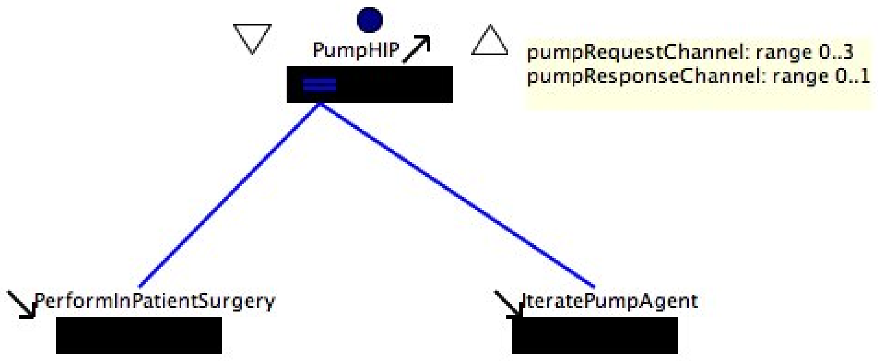
PumpHIP
pumpRequestChannel: range 0..3
pumpResponseChannel: range 0..1
PerformInPatientSurgery
IteratePumpAgent

Figure 4: Pump HIP written in Little-JIL

To illustrate for the motivating example from the introduction, we manually built a new pump HIP model from an existing pump process coordinator model captured by the step `PerformInPatientSurgery` and the newly created pump computational agent model that will be captured by a step `IteratePumpAgent`. Figure 4 shows that new HIP model written in Little-JIL. Non-leaf step `PumpHIP` is the entry point (designated by the northeast arrow to the right of the step name) that declares two sub-steps: a step reference to step `PerformInPatientSurgery` and a step reference to step `IteratePumpAgent`. The two sub-steps will be executed in parallel (designated by the equal sign in the left hand side of the step bar) therefore their subtasks will be interleaved. The step `PumpHIP` allows the human agents to communicate with the pump computational agent by declaring two channels pumpRequestChannel and pumpResponseChannel (designated by the filled in circle above the step bar). In the following for brevity, the channel pumpRequestChannel will be called pumpReqChn and the channel pumpResponseChannel will be called pumpRespChn.

Figure 5 shows the non-leaf step `Configure PCA` that captures how the pump coordinator encodes an RPC to the procedure `configurePCA` of the pump computational agent. The non-leaf step `Configure PCA` has two children `Call configurePCA` and `Return configurePCA` that are executed sequentially (designated by the left arrow in the left hand side of the step bar). First, it encodes the procedure name and in parameters as a request represented as an integer. Next for the call, the first child `Call configurePCA` sends that request to the channel pumpReqChn. Lastly for the return, the second step `Return configurePCA` receives a response from the channel

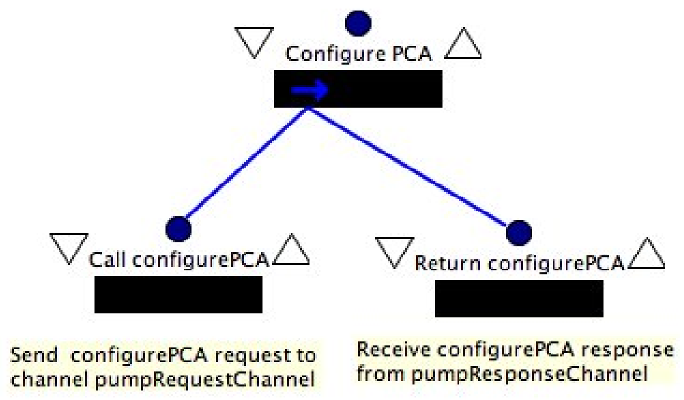
Configure PCA
Call configurePCA
Return configurePCA
Send configurePCA request to channel pumpRequestChannel
Receive configurePCA response from pumpResponseChannel

Figure 5: Pump process coordinator's step `Configure PCA` written in Little-JIL

pumpRespChn and decodes that response into either the out parameters or exceptions thrown.

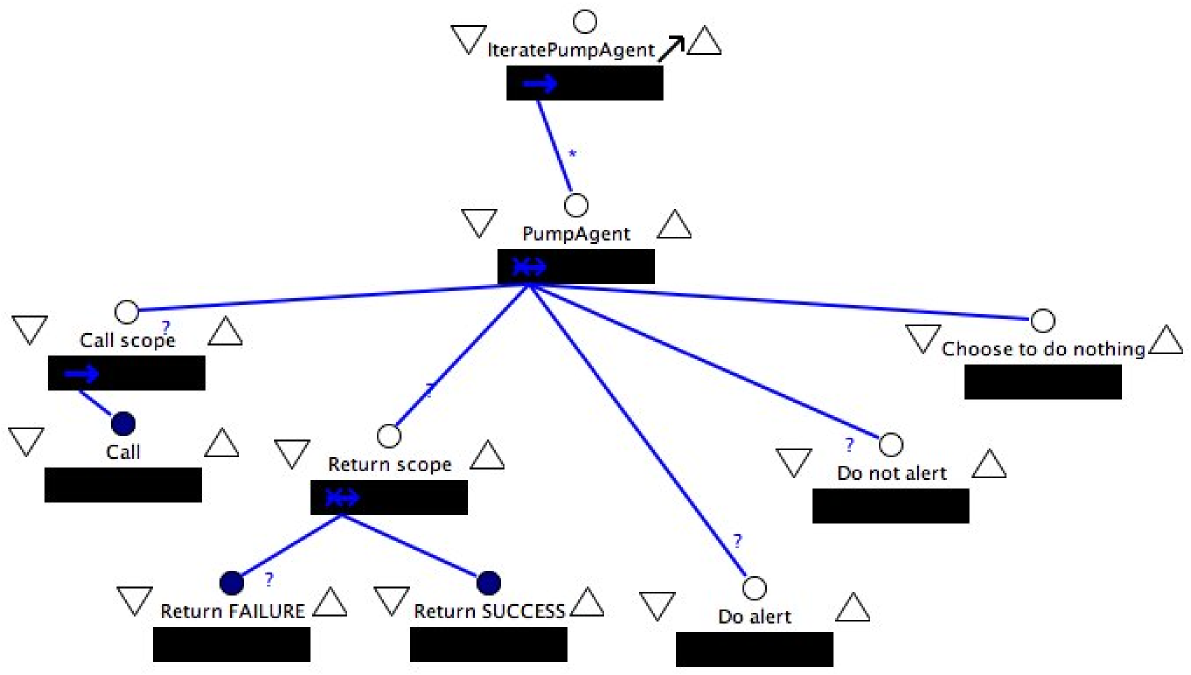

Figure 6: Pump computational agent written in Little-JIL

As described in Section 3, a pessimistic computational agent essentially iterates repeatedly where on each iteration it non-deterministically chooses to execute one of its leaf steps (i.e. procedures) or chooses to do nothing. To illustrate, we manually built the pessimistic pump computational agent model shown in Figure 6 that is written in Little-JIL. The iterations are captured by the kleene star annotating the arc from step `IteratePumpAgent` to sub-step `PumpAgent`. Each iteration is captured by the sub-step `PumpAgent` where the non-deterministically choosing among executing one of its procedures or doing nothing is captured by the sub-step `PumpAgent` being a try step that has optional substeps (designated by the ? annotating the arc from step `PumpAgent` to a sub-step). For the computational agent, a RPC is encoded as a receive from the channel pumpReqChn paired with a send to channel pumpRespChn captured by the sub-step `Call` and the sub-steps `Return FAILURE` and `Return SUCCESS` respectively. The pump computational agent may chose to execute none of its procedures captured by the sub-step `Choose to do nothing`. Lastly, the pump reporting an alert and not reporting an alert are captured by the two sub-steps `Do alert` and `Do not alert` respectively.

4.3.2 FSV model builder tool

The Little-JIL front-end translates from a process model written in Little-JIL to a TFG. It must be provided with event bindings that map the Little-JIL subtasks to the property/constraint events. The supported event bindings are a:

- step state event binding that captures that a named step enters a particular state where the state is one of POSTED, STARTED, COMPLETED, TERMINATED, OPTEDOUT, or RETRACTED
- exception event binding that captures that a named step throws a particular exception where the exceptions are enumerated in the artifact collection
- parameter event binding that captures that a named step accesses a named parameter where the access is either DEFINED or USED

For the requirement derivation toolset, we extended the supported event bindings to include a channel event binding that captures that a named channel is communicated over where the communication is either a SEND (i.e. write) or a RECEIVE (i.e. take).

For the requirement derivation toolset, the Little-JIL front-end was also extended. It now supports channel write with blocking semantics so that when the channel is empty then the write completes immediately but when the channel is full then the write blocks (as opposed to overwrite semantics). Iteration is now translated as a loop in the CFG (as opposed to unrolling the loop a given number of times in the CFG). Integer ranges from a minimum value of 0 to a user-defined maximum value greater than or equal to 0 are translated. Simple predicate expressions where the relational operators (==, !=, $>, >=, <, <=$) are used to compare a named parameter with a constant value (boolean or integer range) are translated. Constant bindings where a named parameter is defined to a constant value (boolean or integer range) are also translated.

4.3.3 Requirement deriver tool

The requirement deriver tool inputs a derivation problem that consists of the HIP process model, its decomposition into module 1 and module 2, the property to be verified, and the set of all constraints to be applied.

The HIP process model is composes of a process coordinator model and a computational agent model. In the HIP FSV model, the HIP, process coordinator, and computational agent are translated as CFGs (i.e. threads). In the HIP process model, the human agents communicate with the computational agent by using RPCs. For a particular computational agent, the RPCs are modeled using a pair of channels, a channel for requests and a channel for responses. In the HIP FSV model, the channels are translated to global variables while the parameters are translated to local variables. Additionally, any exceptions are translated as local variables. For the case studies, the HIP model is always decomposed such that module 1 contains the computational agent model and module 2 contains the process coordinator model.

The set of all constraints were built as follows. FLAVERS/Little-JIL was employed to automatically build the thread related constraints for the threads that correspond to the HIP, process coordinator, and computational agent and to automatically build the variable constraints for the artifacts, including the channels, parameters, and exceptions. For the RPCs, we manually built a data dependency constraint that ensures that the data dependencies among the channels and parameters are represented in the TFG. Lastly, we manually built a computational agent's behavior constraint that encodes any initial requirements on the computational agent's behavior.

The requirement deriver tool implements the extended assumption learning algorithm described in Section 3.2 that combines the L* algorithm and the FLAVERS/Little-JIL FSV tool. It outputs whether or not the HIP satisfies the HIP requirement and a derived requirement of the computational agent's behavior represented as an FSA.Conceptually, the derived requirement of the computational agent partitions the sequences of RPCs to that computational agent into the sequences that satisfy the HIP requirement and the remaining sequences that violate the HIP requirement. To evaluate the requirement derivation approach described in Section 3, we applied the requirement derivation toolset described in this section to two case studies described in the next section.

5 Evaluation

We applied the requirement derivation toolset described in Section 4 to two case studies, a pump case study and an ICD (implantable cardioverter-defibrillator) case study. The requirement derivation toolset is provided as input a HIP model composed of a computational agent model and a process coordinator model written in Little-JIL, a HIP requirement represented as an FSA, and any initial requirements on the computational agent's behavior represented as an FSA. It outputs a derived computational agent requirement represented as an FSA. Section 5.1 describes our investigation of how the requirement derivation approach's inputs, a HIP and a HIP requirement, affect the performance of the requirement derivation in terms of space and time and also the derived requirement that is output. Section 5.2. describes our investigation of how the assumption learning algorithm's optimizations affect the requirement derivation's performance in terms of space and time.

In the introduction, the pump case study was used as the motivating example. In more detail, a pump administers fluids or medications (i.e. drugs) usually intravenously. It typically is used to administer a large set of drugs over a wide range of dosing parameters, often administered through multiple channels. Because of this complexity, infusions lead to such reported medical errors as administration of the wrong drug, administration of an over/under dose, and drug interactions. The pump case study considers a pump HIP composed of a pessimistic computational agent for a pump and a process coordinator for an in-patient surgery in a hospital based on scenarios described in [6]. The pump HIP requirement was taken directly from the safety goals discussed in [6]. We describe the pump case study in more detail in Section 5.1.1.

The second case study is the ICD case study. An ICD "is a device that monitors and responds to heart activity. ICDs have modes for pacing, wherein the device periodically sends a small electrical stimulus to the heart, and for defibrillation, wherein the device sends a larger shock to restore normal heart rhythm" [24]. The ICD case study considers an ICD HIP composed of a pessimistic computational agent for an ICD and a process coordinator for an ICD patient's care. The ICD computational agent model is based on [20, 24]. The ICD process coordinator model is primarily based on the observations by Kevin Fu and his colleagues of an implantation of a new ICD and a battery replacement for an existing ICD performed at a local area medical center [20]. The ICD HIP requirement was taken directly from the security and privacy goals discussed in [23]. We describe the ICD case study in more detail in Section 5.1.2.

The requirement derivation toolset is implemented in Java 1.5. For each case study, we defined one or more derivation problems as described in Section 4.3.3. The requirement derivation toolset was run on each derivation problem to output whether the HIP satisfies the HIP requirement and a derived computational agent requirement represented as an FSA. The runs were space bounded (i.e. Java had its maximum heap size set to 2 GB) but not time bounded (i.e. ran to completion when the derived requirement was output or ran to termination when an OutOfMemoryError was thrown). For each run, we measured the requirement derivation toolset's performance in terms of both space and time.

The experimental platform is a MacBook Pro running Mac OS X version 10.5.8 with a 2.4 GHz Intel Core 2 Duo and 4 GB 667 MHz DDR2 SDRAM. The Java virtual machine (JVM) was running the Java 2 Runtime Environment, Standard Edition (build 1.5.0_22-b03-333-9M3125) where the JVM only supports a maximum heap size of a little over 2 GB (by trial and error around 2100MB). The derivation space was measured using the Java `Runtime` class, specifically we use the `totalMemory` method and the `freeMemory` method. It was calculated as the difference between the total and free memory. The Java Runtime class is known to report a very crude measure of memory usage. The derivation time was measured using the Java `Date` class, specifically we used the `getTime` method. We recorded the start Date before beginning the derivation and the finish Date after ending the derivation. The derivation time was calculated as the difference between the finish time and the start time.

5.1 How the requirement derivation approach's inputs, a HIP and a HIP requirement, affect the performance of the requirement derivation in terms of space and time and also the derived requirement that is output

Our goal was to evaluate whether or not the automated requirement derivation approach:

1. can be applied to "real" HIPs (i.e. what is the performance in terms of space and time)
2. can be applied to safety, security, and privacy properties and, if possible, whether it is useful to do so
3. can derive requirements of the computational agent that are readily understandable and useful to computational agent developers (i.e. fill in missed or correct inaccurate requirements of the computational agent)

The next two sections discuss the pump case study and the ICD case study.

5.1.1 Pump Case Study

The "smart" pumps typically load drug libraries where each drug library is associated with a particular primary care area (PCA) and enumerates the drugs in use in that PCA. Further each drug in use is associated with the usual dosing parameters such as concentrations, dosing units, and dosing limits (specifically a minimum dosing limit and a maximum dosing limit). Before being used, a pump is now configured to a particular drug library by selecting a PCA. This allows a pump to check inputs (e.g. check that the drug may be used in that PCA, that the dose is within limits, or that there is no drug interaction) and alert when those checks fail. But as mentioned in the introduction, the pumps are used in different primary care areas such as the operation room and the intensive care unit by various human agents such as anesthesiologists and nurses. Therefore, it can be challenging to ascertain whether or not the pump is used correctly to help ensure the patient's safety.

Pump HIP For the pump HIP model, the resource repository defines human agents (e.g. a surgeon, nurse, clerk, and patient) and a computational agent for a pump. The artifact collection defines artifacts such as patient identifier (ID) bracelets, medical records, and surgical tools. The coordination specification for the HIP is composed of a computational agent model for a pump and a process coordinator model that defines an in-patient surgery in a hospital. In general within the coordination specifications, we elaborated those steps where the human agents communicated with the pump computational agent. Overall, the pump HIP model written in Little-JIL contains 34 steps. The number of steps is defined to be all of the steps reachable from the entry point step of the HIP model where each step reference is resolved to its step declaration. In the following, we describe the pump computational agent first and the pump process coordinator model second.

Pump computational agent For the pump computational agent model, we made several simplifying assumptions. There are only two drug libraries modeled: the OR's drug library and the ICU's drug library. For each PCA's drug library, there is a single drug modeled where the only dosing parameters modeled are the dosing limits, specifically a minimum dosing limit and a maximum dosing limit. The drug doses are abstracted as either a lower dose or a higher dose. The API for the pump computational agent model is:

- type PrimaryCareArea is enumerated (OR, ICU)
- type Dose is enumerated (LOWERDOSE, HIGHERDOSE)
- type ResponseCode is enumerated (SUCCESS, FAILURE)
- procedure `configurePCA` (pca: PrimaryCareArea) returns ResponseCode
- procedure `enterDose` (ds: Dose) returns ResponseCode

The pump computational agent model was described in Section 4.3.1. As written in Little-JIL, it contains 10 steps.

Pump process coordinator The pump process coordinator model has as its entry point the step `PerformInPatientSurgery` that captures the major stages of an in-patient surgery in a hospital at a high level of abstraction. It sequentially executes the sub-steps `Check-in patien`, `Perform operation`, `Administer ICU care`, `Administer post-operative care`, `Check-out patient`. We focus on the sub-steps `Perform operation` and `Administer ICU care` since they may communicate with the pump computational agent, specifically they both contain a step reference to the sub-step `Perform infusion` described in the next paragraph.

Figure 7 shows the step `Perform infusion` that sequentially executes the sub-steps `Configure PCA`, `Enter dose`, and `Administer infusion`. The sub-step `Configure PCA` is optional (designated by the ? annotating the incoming arc). The sub-step `Enter dose` may throw a DoseExceedsLimit exception that is propagated to the step `Perform infusion`. The exception handler `Respond to dose exceeds limit alert` (designated by the X in the right hand side of the step bar) catches the DoseExceedsLimit exception and handles it by completing the step `Perform infusion`. This ensures that if a health care practitioner entered a dose that exceeds the dosing limits then that health care practitioner does not administer the over/under dose to the patient. In other words, the exceptional control flow ensured that the sub-step `Administer infusion` was not executed. For space considerations, the remainder of the pump process coordinator is now shown. Overall, it contains 23 steps.

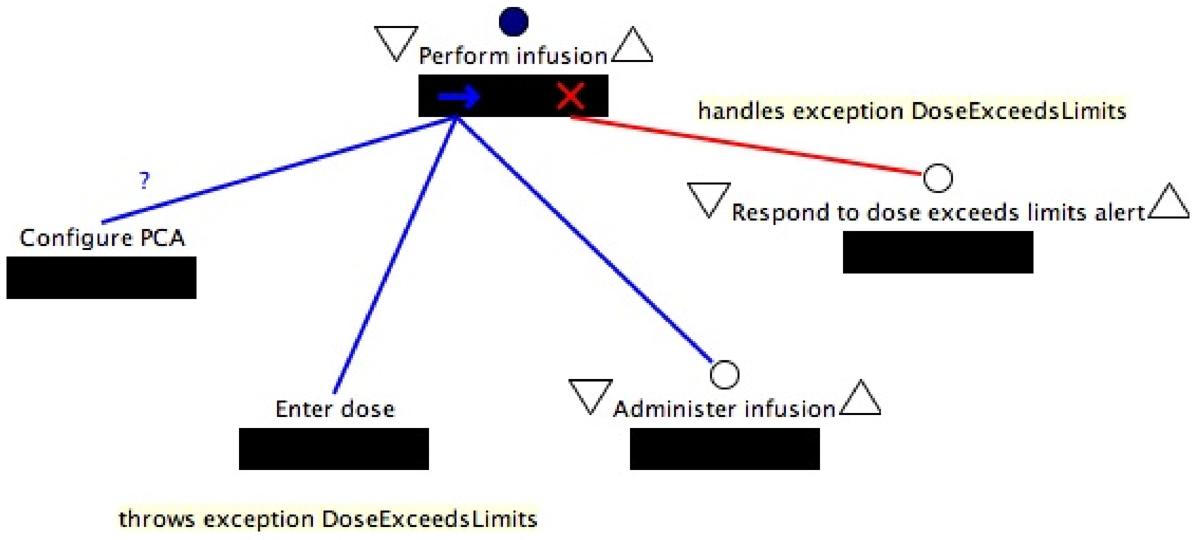
Perform infusion
handles exception DoseExceedsLimits
?
Configure PCA
Respond to dose exceeds limits alert
Enter dose
Administer infusion
throws exception DoseExceedsLimits

Figure 7: The step `Perform infusion` written in Little-JIL

Pump Property The pump HIP requirement considers safety. Informally, it states that inside the ICU if a health care practitioner enters a higher dose then the pump alerts that the entered dose exceeds the dosing limits. For the pump HIP requirement, the alphabet contains the events *EnterICU*, *LeaveICU*, *HigherDose*, *PumpAlert*, and *NotPumpAlert*. Figure 8 shows the pump HIP requirement as an FSA. In the figure, each state is depicted as a circle. The start state, state 1, is annotated with an arrow. An accepting state, e.g. state 3, is depicted as two concentric circles. Each transition is depicted as an arc between a source state and a target state annotated with a list of events from the alphabet. For example, there exists the self loop transition from source state 1 to target state 1 on the list of events *HigherDose*, *PumpAlert*, *NotPumpAlert*. Each FSA was made total by adding a violation state that is a non-accepting state and also a sink state meaning that all its transitions are self loops. For space considerations, the figure does not show the violation state or its transitions. Therefore in the figure, if a given state explicitly does not have a transition for a particular event in the alphabet that is shown, then that state implicitly has a transition for that event to the violation state that is not shown. To illustrate, there exists an implicit transition from the source state 1 on the event *LeaveICU* to the violation state that is not shown.

In Figure 8, the start state, state 1, represents being outside the ICU. State 1's self loop transitions represent that outside the ICU if a health care practitioner enters a higher dose then the pump may or may not alert the health care practitioner. States 2 and 3 represent being inside the ICU. The transition from state 1 on event *EnterICU* to state 2 represents moving from outside the ICU to inside the ICU while the transition from state 2 on event *LeaveICU* to state 1 represents moving from inside the ICU to outside the ICU. The transition from state 2 on event *HigherDose* to state 3 represents that the health care practitioner entered a higher dose. The transition from state 3 on event *PumpAlert* to state 2 represents that the pump did alert the health care practitioner about exceeding the dosing limits while the transition from state 3 on event *NotPumpAlert* to the violation state represents that the pump did not alert the health care practitioner about exceeding the dosing limits.

For FLAVERS/Little-JIL, the event *EnterICU* is bound to the process coordinator's step

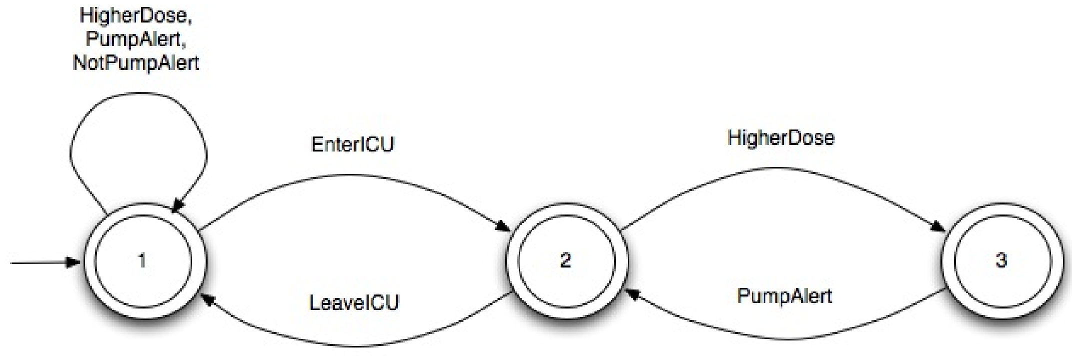

Figure 8: Pump property as an FSA

`Administer ICU care` entering the STARTING state. The event *LeaveICU* is bound to the process coordinator's step `Adminster ICU care` entering the COMPLETING state. The event *HigherDose* is bound to the process coordinator's step `Call enterDose(HIGHERDOSE)` sending 3 to the pumpReqChn channel. The event *PumpAlert* is bound to the agent's step `Do alert` entering the COMPLETING state. The event *NotPumpAlert* is bound to the agent's step `Do not alert` entering the COMPLETING state. Overall, the property is not satisfied if the property FSA reaches the violation state.

Pump Requirement Derivation For the pump case study, the requirement derivation toolset was given the pump HIP model composed of the pump computational agent model described in Section 5.1.1 and the pump process coordinator model shown in Figure 22, the pump HIP property shown in Figure 8, and no initial requirements on the pump computational agent's behavior. The requirement derivation toolset output that the pump HIP violates the safety HIP requirement and a derived requirement of the pump computational agent. The pump HIP model violates the HIP requirement because the pump computational agent model is too pessimistic. In particular, if the pump process coordinator calls the procedure `enterDose` with higher dose then the pump computational agent non-deteriministically chooses between reporting an alert and not reporting an alert. The derived requirement of the pump computational agent was readily understandable since it states that outside the ICU the pump can perform any of its procedures while inside the ICU the pump cannot perform any of its procedures. But, the derived requirement was too strong to be useful. A pump HIP where the pump computational agent satisfies the derived requirement would vacuously satisfy the pump HIP requirement since the pump computational agent can perform none of its procedures. Specifically, the pump computational agent cannot perform the procedure `enterDose` given a higher dose and then not report an alert. So next, the requirement derivation toolset was provided with initial requirements on the pump computational agent's behavior in an effort to make the derived requirement more useful.

Informally, the initial requirements on the pump computational agent are that the pump stores the configured PCA (i.e. which PCA it was configured for last). Initially, the pump computational agent is configured with the ICU's drug library. The procedure `configurePCA`(pca: PrimaryCareArea) updates the configured PCA to be the given PCA. Additionally for the configured PCA's drug library, the pump stores its dosing limits, specifically the minimum dose and the maximum dose. In the ICU's drug library, the minimum/maximum dose is lower dose. But in the OR's drug library, the minimum dose is lower dose and the maximum dose is higher dose. The

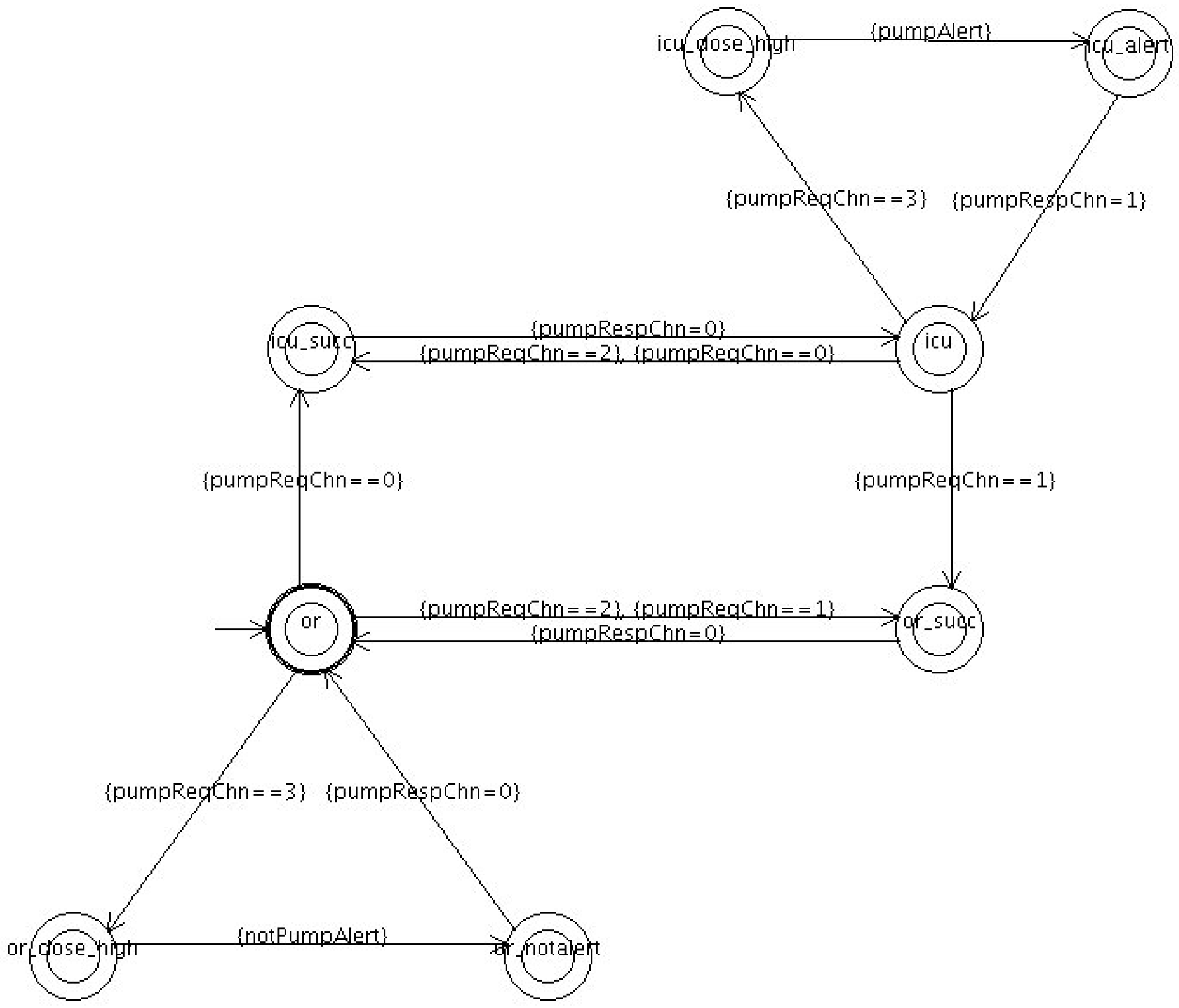

Figure 9: The pump computational agent's behavior constraint as an FSA

procedure `enterDose`(dose: Dose) checks whether the given dose exceeds the configured PCA's minimum or maximum dose. If so, then the pump does alert that the given dose exceeds the dosing limits. If not, then the pump does not alert. Formally, the initial requirements on the pump computational agent's behavior were captured as a constraint. Since the initial requirements of the pump computational agent primarily involved the pump's procedures, the constraint encodes the RPCs to the pump's procedures as integer ranges. The channel pumpReqChn has an integer range type from 0 to 3 inclusive where call `configurePCA(ICU)` is 0, call `configurePCA(OR)` is 1, call `enterDose(LOWERDOSE)` is 2, call `enterDose(HIGHERDOSE)` is 3. The channel pumpRespChn has an integer range type from 0 to 1 inclusive where return SUCCESS is 0 and return FAILURE is 1.

Figure 9 shows the pump computational agent's behavior constraint as an FSA. Initially, the pump is configured with the ICU's drug library represented by the start state, state 'icu'. A call to procedure `configurePCA` with the pca parameter set to ICU and returning a response code of SUCCESS is represented by the transition from state 'icu' on event *pumpReqChn==0* to state 'icu_succ' and then the transition from state 'icu_succ' on event *pumpRespChn=0* to state 'icu'. A call to procedure `enterDose` with the dose set to LOWERDOSE and returning a response code of SUCCESS is represented by the transition from state 'icu' on event *pumpReqChn==2* to state 'icu_succ' and then the transition from state 'icu_succ' on event *pumpRespChn=0* to state 'icu'. A call to procedure `enterDose` with the dose set to HIGHERDOSE, alerting that the dose exceeds limits, and then returning a response code of FAILURE is represented by the transition from state 'icu' on event *pumpReqChn==3* to state 'icu_check', the transition from state 'icu_check' on event *PumpAlert* to state 'icu_alert', and the transition from state 'icu_alert' on event *pumpRespChn=1* to state 'icu'.

When the pump is configured with the OR's drug library, it is represented by state 'or'. A call to procedure `configurePCA` with the pca parameter set to OR and returning a response code of SUCCESS is represented by the transition from state 'icu' on event *pumpReqChn==1* to state 'or_succ' and then the transition from state 'or_succ' on event *pumpRespChn=0* to state 'or'. For the OR, the pump agent's behavior is similar to the ICU with one key difference. That difference is that a call to procedure `enterDose` with the dose set to HIGHERDOSE does not alert and returns a response code of SUCCESS represented by the transition from state 'or' on event *pumpReqChn==3* to state 'or_check', the transition from state 'or_check' on event *NotPumpAlert* to state 'or_notalert', and the transition from state 'or_notalert' on event *pumpRespChn=0* to state 'or'.

For the pump case study, the requirement derivation toolset was given the pump HIP model composed of the pump computational agent model described in Section 5.1.1 and the pump process coordinator model shown in Figure 22, the pump HIP property shown in Figure 8, and *the pump computational agent's behavior constraint* shown in Figure 9. The requirement derivation toolset output that the pump HIP violates the safety HIP requirement and a derived requirement of the pump computational agent. The pump HIP model violates the HIP requirement because the health care practitioners do not always configure the pump for a particular primary care area before they administer infusions in that area. Informally, the derived requirement of the pump computational agent states that after the pump is moved into the ICU that pump must be reconfigured for the ICU before that pump is used to administer infusions. But for the requirement derivation toolset, the derivation space needed was about 115 MB and the derivation time needed was over an hour. Additionally since the derived requirement of the pump computational agent contains 46 states, it was not readily understandable. Figure 10 shows the derived requirement of the pump computational agent as an FSA. The derived requirements represented as FSAs were visualized using the Graphviz tool [21] primarily developed at AT&T Research. In the figures of the derived requirements represented as FSAs, the start state is depicted as a rectangle and the remaining states are depicted as circles. Each accepting state is depicted by being shaded. The violation state is not shown.

When we investigated the requirement derivation toolset's poor performance and the large size of the derived requirement, we ascertained that the requirement derivation toolset considered many infeasible paths. Specifically within the HIP FSV model the TFG and constraints capture the RPC semantics with regards to the channel communications and the parameters that store the requests/responses. But it does not capture the RPC semantics with regards to an RPC's call and return being paired. The infeasible paths impacted the performance negatively by increasing the space and time used and made the derived requirement less understandable by increasing its size. Thus, FLAVERS/Little-JIL was extended to remove such infeasible paths from consideration. We provided RPC constraints that capture the RPC semantics with regard to the call/return pairs. To illustrate, we consider the human agents that communicate with the pump computational agent over the two channels pumpReqChn and pumpRespChn. The channel pumpReqChn contains requests that range from 0 to 3 inclusive and the channel pumpRespChn contains responses that range from 0 to 1 inclusive.

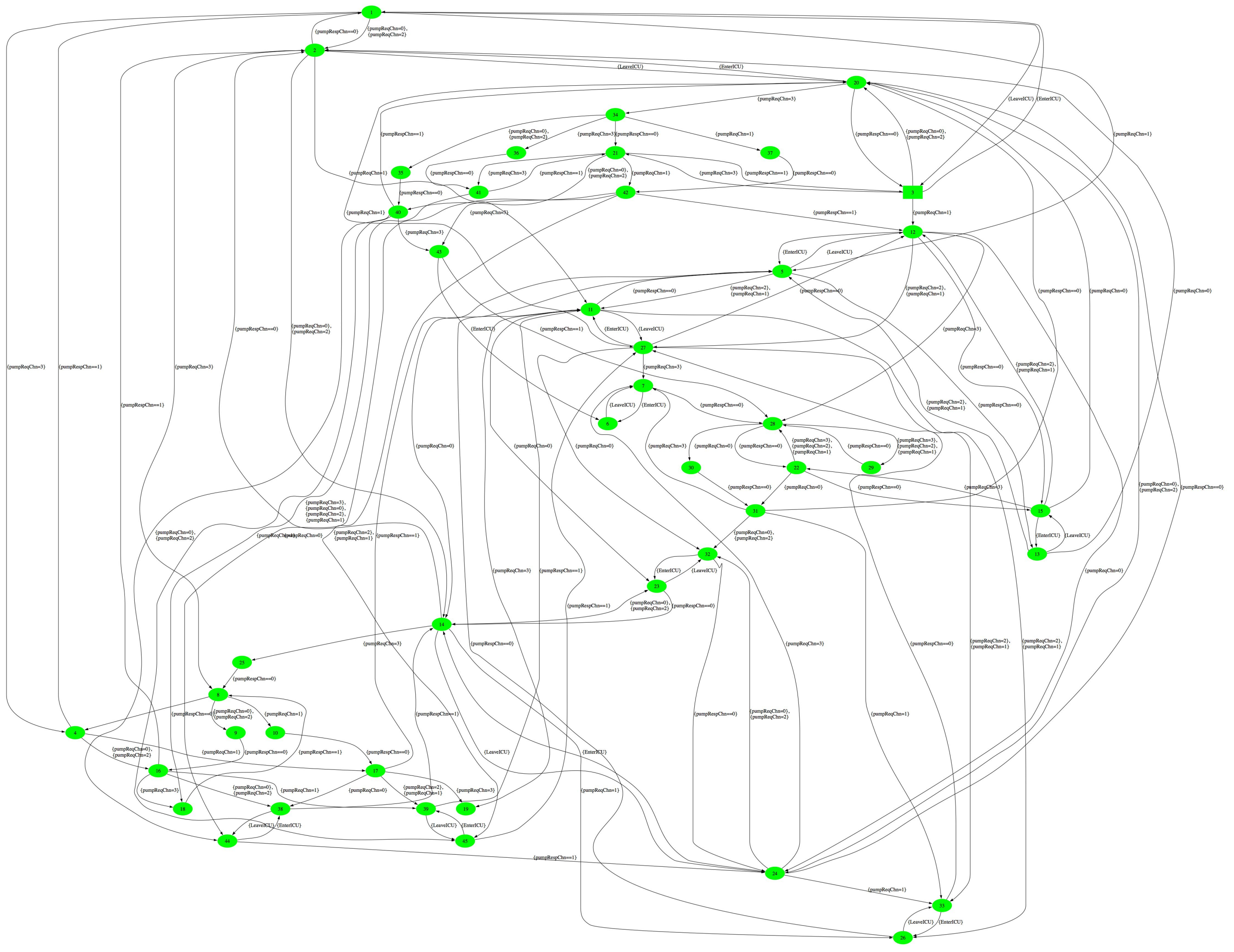

Figure 10: Derived requirement of the pump computational agent as an FSA

Figure 11 shows the pump process coordinator's RPC constraint as an FSA that captures the process coordinator's side of the RPC. Initially, the pump process coordinator is at the start state, state 'coord before call'. The pump process coordinator begins an RPC by calling a particular procedure with given in parameters encoded as request X. This is represented by the transition from state 'coord before call' on event *pumpReqChn=X* to state 'coord before return'. Next, the pump process coordinator waits for the pump computational agent to execute that procedure on those in parameters represented by state 'coord before return'. Lastly, the pump process coordinator ends the RPC for that procedure by accepting the return of the out parameters or exceptions thrown encoded as response Y. This is represented by the transition from state 'coord before return' on event *pumpRespChn==Y* to state 'coord before call'.

Figure 12 shows the pump computational agent's RPC constraint as an FSA that captures the computational agent's side of the RPC. Initially, the pump computational agent is at the start state, state 'agent before call'. The pump computational agent begins an RPC by accepting a call with given in parameters encoded as a request X. This is represented by the transition from state 'agent

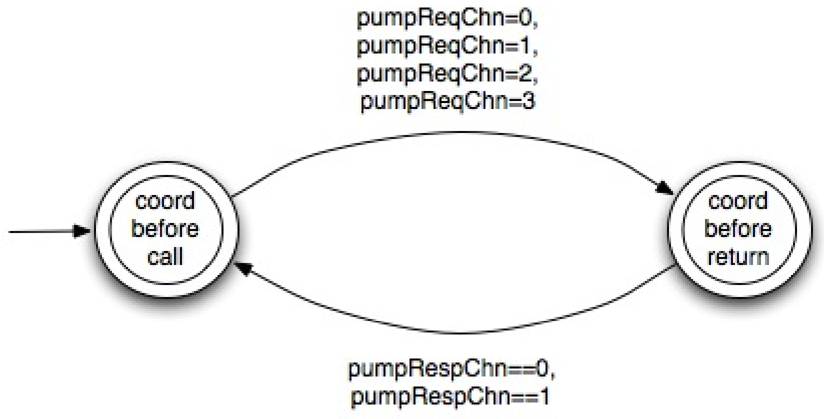

Figure 11: RPC constraint for the pump process coordinator as an FSA where the pump process coordinator communicates over channels pumpReqChn and pumpRespChn

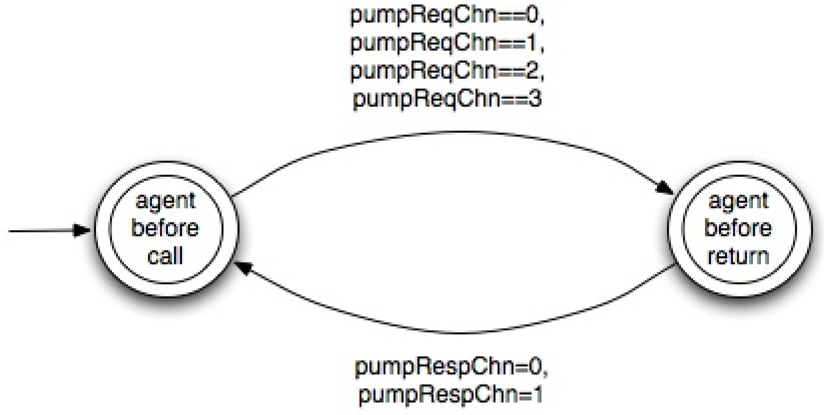

Figure 12: RPC constraint for the pump computational agent as an FSA where the pump computational agent communicates over channels pumpReqChn and pumpRespChn

before call' on event *pumpReqChn==X* to state 'agent before return'. Next, the pump computational agent executes the procedure on the given in parameters that corresponds to request X. This is represented by state 'agent before return'. After the requested procedure completes/terminates, the pump computational agent ends the RPC for that procedure by returning the out parameters or exceptions thrown encoded as response Y. This is represented by the transition from state 'agent before return' on event *pumpRespChn=Y* to state 'agent before call'.

Now for the pump case study, the derivation problem was changed by adding the RPC constraint for the pump process coordinator and the RPC constraint for the pump computational agent. The requirement derivation toolset was rerun and took 34 MB and 245 seconds. Figure 13 shows the derived requirement of the computational agent represented as an FSA that now contains 12 states. So, the RPC constraints improved the requirement derivation's performance and the understandability of the derived requirement. But, the derived requirements could become even more understandable if the complexity introduced by the RPCs was abstracted away. Conceptually, we abstract an RPC for a particular procedure by pairing its call to that procedure and its return from that procedure (e.g. events *pumpReqChn=0*, *pumpRespChn==0*) and replacing that pair with the step that corresponds to that procedure (e.g. *pump_configurePCA_ICU_succ*).

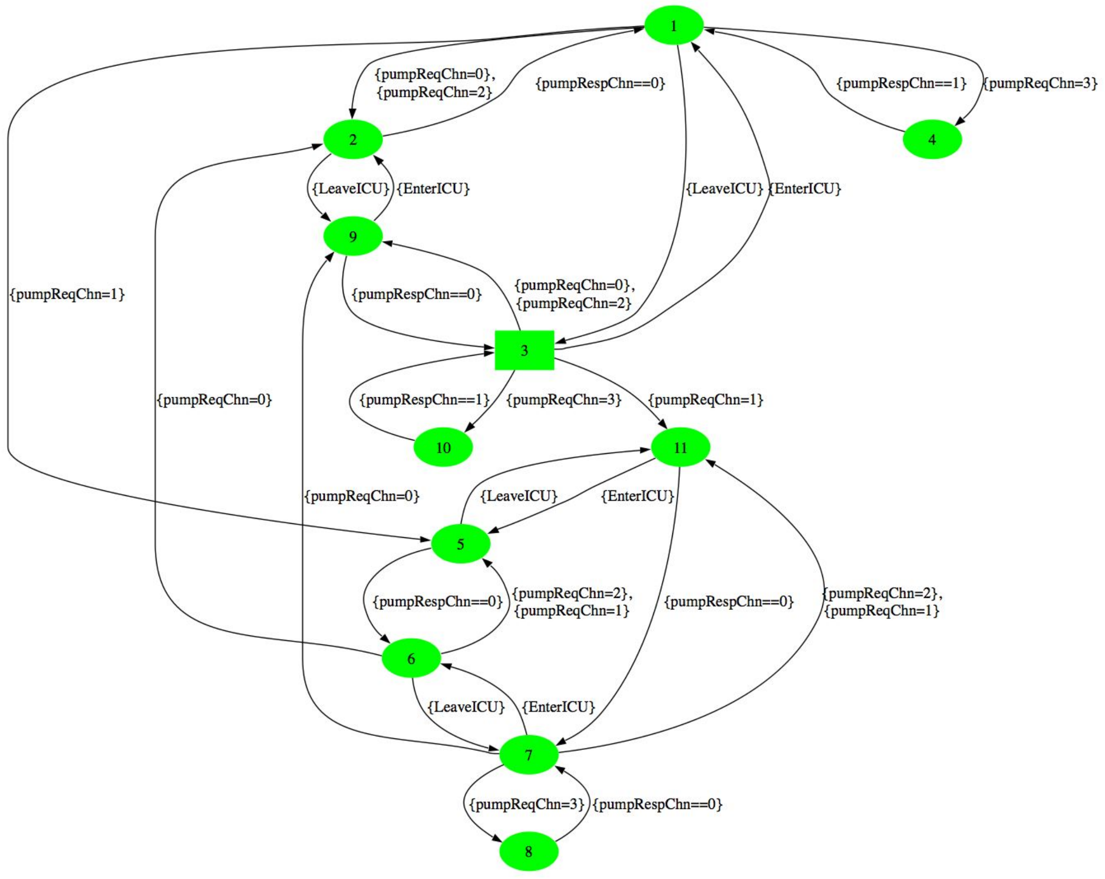

Figure 13: Derived requirement of the pump computational agent as an FSA with initial requirements of the pump computational agent

We built a derived requirement abstraction tool that inputs a derived requirement represented in the RPC view and outputs that derived requirement represented in the step view. At a high-level, the derived requirement abstraction tool partitions the derived requirement's alphabet into the RPC related events (e.g. events *pumpReqChn=0*, *pumpRespChn==0*) and the other events (e.g. *EnterICU*). For now, the user provides a map from each RPC related event pair to a step event (e.g. RPC related event *pumpReqChn=0* paired with RPC related event *pumpRespChn==0* maps to step event *pump_configurePCA_ICU_succ*). The derived requirement is converted from the RPC view to the step view using two patterns:

- RPC pattern: Input a transition from old state S1 on old event e1 to old state S2 and another transition from old state S2 on old event e2 to old state S3; Output if (event pair e1,e2 maps to step event e) then create a new transition from new state S1 on step event e to new state S3

- Other pattern: Inputs a transition from old state S1 on old event e1 to old state S2; Output if (other events contains e) then create a new transition from new state S1 on new event e1 to new state S2

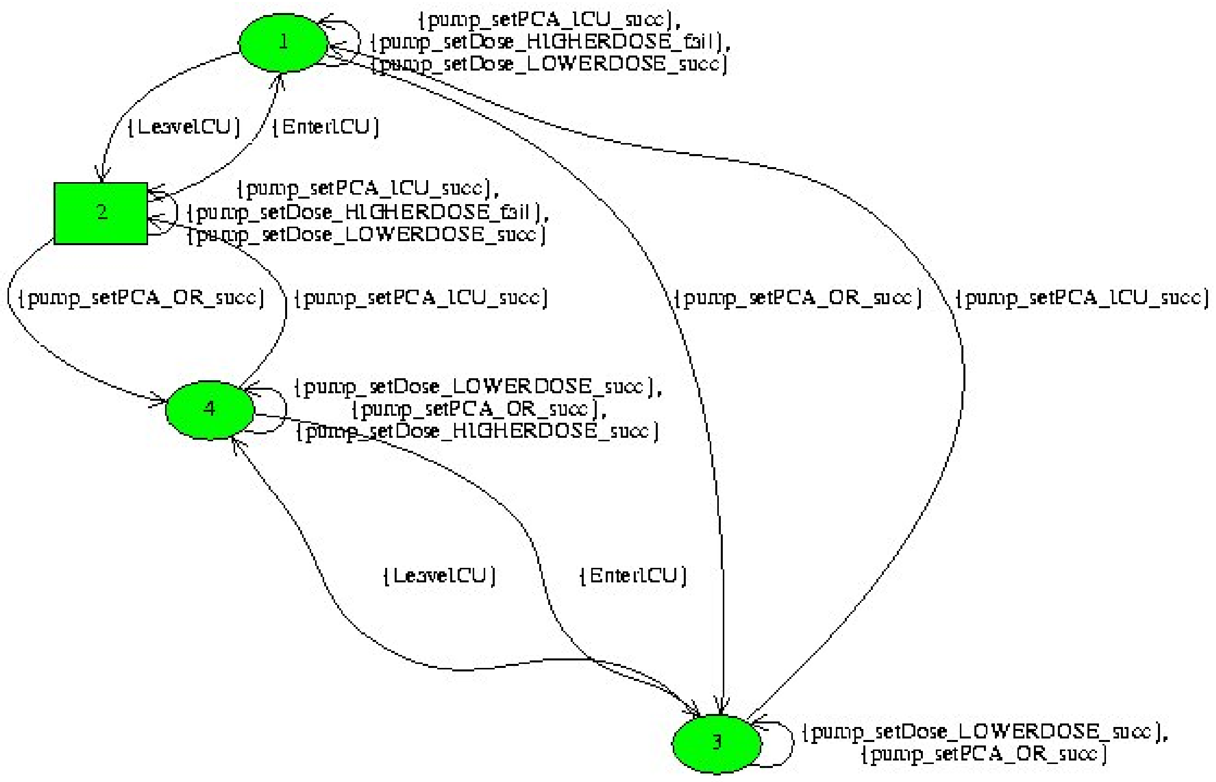

Figure 14: Derived requirement of the pump computational agent as an FSA

We describe the derived requirement abstraction tool's algorithm in more detail in the Appendix.

Figure 14 shows the step view of the pump derived requirement where the RPC related event pairs map to the step events as follows:

- event *pumpRequestChannel=0* paired with event *pumpResponseChannel==0* maps to event *pump_configurePCA_ICU_succ*
- event *pumpRequestChannel=0* paired with event *pumpResponseChannel==1* maps to event *pump_configurePCA_ICU_fail*
- event *pumpRequestChannel=1* paired with event *pumpResponseChannel==0* maps to event *pump_configurePCA_OR_succ*
- event *pumpRequestChannel=1* paired with event *pumpResponseChannel==1* maps to event *pump_configurePCA_OR_fail*
- event *pumpRequestChannel=2* paired with event *pumpResponseChannel==0* maps to event *pump_enterDose_LOWERDOSE_succ*
- event *pumpRequestChannel=2* paired with event *pumpResponseChannel==1* maps to event *pump_enterDose_LOWERDOSE_fail*
- event *pumpRequestChannel=3* paired with event *pumpResponseChannel==0* maps to event *pump_enterDose_HIGHERDOSE_succ*

- event *pumpRequestChannel=3* paired with event *pumpResponseChannel==1* (i.e. entering a higher dose leads to a pump alert) maps to event *pump_enterDose_HIGHERDOSE_fail*

Overall, the derived requirement is now more understandable. The pump computational agent derived requirement highlights that a pump in a particular PCA must be configured for that PCA before being used in that PCA to administer infusions. This is primarily captured by the event sequence *pump_configurePCA_OR_succ,EnterICU, pump_configurePCA_ICU_succ, pump_enterDose_HIGHERDOSE_fail, LeaveICU.* Next, we consider whether or not the HIP requirement considering safety is useful and whether or not the requirement derivation toolset's output is useful.

The requirement derivation toolset outputs that the pump HIP violates its safety HIP requirement and also the pump computational agent derived requirement shown in Figure 14. For the purpose of preventing medical safety errors, it is useful to know that the pump HIP violates the pump HIP requirement. Specifically to prevent medical safety errors, the pump developers could modify the pump to implement the pump derived requirement. For example, they could make the pump be location sensitive (e.g. query a central computer in the hospital or employ radio-frequency identification tags) so that it can ascertain which PCA it's located in and then the pump could configure itself for that PCA. This would help to ensure that more pump HIPs satisfy the pump HIP requirement about a higher dose leading to a pump alert since the pump computational agent satisfies the pump derived requirement.

To investigate whether the requirement derivation approach can be applied to "real" HIPs, we compared the process coordinator described above with a more detailed version. For the more detailed version, we elaborated the process coordinator by adding references to steps declared in the verify patient identification (ID) process described in [12]. Specifically, the steps `Check-in patient` and `Perform operation` were elaborated. The step `Perform check-in` has a child step `obtain new ID band` that is a leaf step. The leaf step `obtain new ID band` was changed to be a reference to the step `obtain new ID band` declared in the verify patient ID process. Non-leaf step `obtain new ID band` involves a clerk verifying a patient's ID and then placing an ID band on that patient containing her or his identifying information such as name and data of birth. Also, the step `Perform operation` has a child step `verify patient ID band` that is a leaf step. The leaf step `verify patient ID` was changed to be a reference to the step `verify patient ID band` declared in the verify patient ID process. Non-leaf step `verify patient ID band` involves a health care practitioner verifying that a patient's identifying information matches the ID band's identifying information. The pump process coordinator model that is more detailed contains 64 steps. For the pump process coordinator model that is more detailed, the requirement derivation toolset output exactly the same derived requirement of the pump computational agent as for the pump process coordinator model that is less detailed. That derived requirement is shown in Figure 14.

For the requirement derivation toolset runs, we partitioned the measurements by the requirement derivation, the L* algorithm, and FLAVERS/Little-JIL. For the requirement derivation, we measured the number of steps in the HIP written in Little-JIL (Little-JIL step count), the number of states in the derived FSA (state count), the space in megabytes (MB) for the derivation (derivation space), and the time in seconds (sec) for the derivation (derivation time). For the L* algorithm, we measured the number of events in the alphabet of the derived FSA (alphabet size), the number of membership queries answered by the L* teacher (membership query count), and the number of equivalence queries answered by the L* teacher (equivalence query count). For FLAVERS/Little-JIL, we measured the number of nodes in the TFG (TFG node count), the number of constraints applied (constraint count), the number of node-tuples stored while answering the last equivalence query for the L* teacher (last node-tuple count), and the sum of all node-tuples ever stored while answering all of the queries for the L* teacher (sum node-tuple count). Since the automated derivation toolset decomposes each HIP into module 1 that contains the computational agent and module 2 that contains the process coordinator, all of the FLAVERS/Little-JIL measurements, except for the sum, are broken into a module 1 component (M1) and a module 2 component (M2) displayed as M1 / M2.

VARIABLE NAME	Pump less detailed	Pump more detailed
Requirement derivation		
Little-JIL step count	34	75
State count	12	12
Derivation space (MB)	44	40
Derivation time (sec)	245	249
L* algorithm		
Alphabet size	8	8
Member. query count	1168	1168
Equiv. query count	10	10
FLAVERS/Little-JIL		
TFG node count	174 / 233	174 / 248
Constraint count	7 / 9	7 / 9
Last node-tuple count	138229 / 54893	138229 / 55326
Sum node-tuple count	6106459	6208842

Table 1: Data collected for the pump case study requirement derivation runs

Table 2 shows the the data collected for the pump case study requirement derivation toolset runs. For both pump HIPs, the derivation space needed was less than 44 MB and the derivation time needed was under 5 minutes. For the pump case study, the requirement derivation toolset can be applied to HIPs that are defined at a high-level of abstraction. In addition, the HIP can be further elaborated with regards to the subtasks that do not communicate with the computational agent without adversely affecting the requirement derivation toolset's performance and also without affecting the derived requirement.

5.1.2 ICD Case Study

From [24], ICDs are being widely adopted. But since multiple human agents such as health care practitioners and patients are coordinating the patients' care with the ICDs in diverse settings such as health care facilities, clinics, and homes, it is challenging to reason about whether the ICDs are being used in a manner that ensures security and privacy. An additional challenge is introduced by human agents such as attackers who may adversely affect the patients' care with the ICDs by intentionally causing the ICDs to fail.

ICD HIP For the ICD HIP model, the resource repository defines human agents such as a surgical team (two cardiologists, three nurses), clerks, patients, attackers and computational agents such as a fluoroscope, an external programmer, and an ICD. We primarily focus on the ICD computational agent. The artifact collection defines artifacts such as patient ID bracelets, medical records, and surgical tools. The coordination specification for the HIP is composed of an ICD computational agent and an ICD process coordinator for the implantation of the ICD into the patient in the hospital proceeded by either a follow up in the clinic or an attack from outside. In general within the coordination specifications, we elaborated those steps where the human agents communicated with the ICD computational agent. Overall, the ICD HIP model written in Little-JIL contains 66 steps. In the following, we describe the ICD computational agent first and the ICD process coordinator second.

ICD computational agent An ICD is implanted in the patient's chest cavity and its connected to the patient's heart with electrical leads. Once the ICD is implanted, an external programmer may be employed to wirelessly access and program the ICD by sending radio commands from the programmer to the ICD. Initially, the ICD rejects all commands from the ICD programmer. A wand is employed to send the command to `activate` the ICD so that it accepts all commands from the external programmer. The wand may use a magnet or a radio signal for the activation. Additionally, we consider the command to `deactivate` the ICD so that it rejects all commands. The command `readdata` reads the ICD's data including the patient's telemetry (i.e. vital sign information) and the ICD's therapies where a therapy describes what telemetry triggers administration of pacing or a shock to restart the heart. The command `testmode` administers a shock to stop the heart. The API for the ICD computational agent model is:

- type ResponseCode is enumerated (SUCCESS,FAILURE)
- procedure `activate`() returns ResponseCode
- procedure `deactivate`() returns ResponseCode
- procedure `readdata`() returns ResponseCode
- procedure `testmode`() returns ResponseCode

The initial requirements on the ICD computational agent's behavior are captured as a constraint. As for the pump case study, the initial requirements on the ICD computational agent primarily involved the ICD's procedures. So the constraint encoded the RPCs to those procedures as integer ranges. The channel icdReqChn has an integer range type from 0 to 3 inclusive where call `activate`() is 0, call `deactivate`() is 1 call `readdata`() is 2, call `testmode`() is 3. The channel icdRespChn has an integer range type from 0 to 1 inclusive where return SUCCESS is 0 and return FAILURE is 1. Figure 15 shows the ICD agent's behavior constraint as an FSA. Initially, the ICD rejects all commands represented by the start state, state 'notAct'. A call to procedure `readdata`, `testmode`, or `deactivate` returns a response code of FAILURE represented by the transition from state 'notAct' on event *icdReqChn==1*, *icdReqChn==2*, *icdReqChn==3* to state 'notAct_reqOther' and then the transition from state 'notAct_reqOther' on event *icdRespChn=1* to state 'notAct'. A call to procedure `activate` returns a response code of SUCCESS represented by the transition from state 'notAct' on event *icdReqChn==0* to state 'notAct_reqAct' and then the transition from state 'notAct_reqAct' on event *icdRespChn=0* to state 'act'.

Now, the ICD accepts all commands represented by the state 'act'. A call to procedure `activate` or `readdata` returns a response code of SUCCESS represented by the transition from state 'act' on event *icdReqChn==0* or *icdReqChn==2* to state 'act_reqOther' and the transition from state 'act_reqOther' on event *icdRespChn=0* to state 'act'. A call to procedure `testmode`, administering the shock, and returning a response code of SUCCESS is represented by the transition from state 'act' on event *icdReqChn==3* to state 'act_reqTM', the transition from state 'act_reqTM' on event *TestMode* to state 'act_TM', and the transition from state 'act_TM' on event *icdRespChn=0* to state 'act'. Finally, a call to procedure `deactivate` returns a response code of SUCCESS represented by the transition from state 'act' on event *icdReqChn==1* to state 'act_reqDeact' and then the

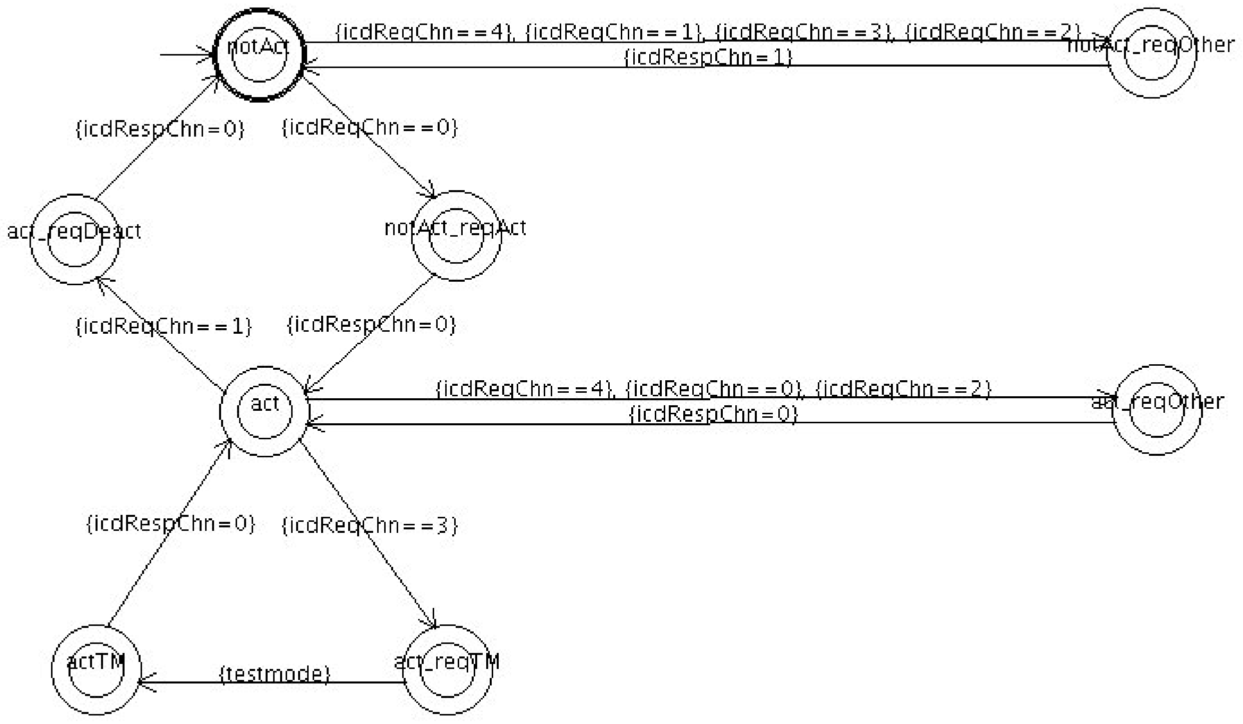

Figure 15: The ICD computational agent's behavior constraint as an FSA

transition from state 'act_reqDeact' on event *icdRespChn=0* to state 'notAct'. As was initially the case, the ICD once again rejects all commands.

The ICD computational agent model was created as described in Section 4.3.1. The ICD computational agent model written in Little-JIL contains 9 steps. It is contained in the Appendix.

ICD process coordinator For the ICD process coordinator model, the major phases are the implantation of the ICD in the patient in the hospital, a follow up in the clinic, or an attack from outside. We focus on the sub-step `Perform implantation` that communicates with the ICD computational agent. The sub-step `Perform follow up` behaves similarly to the sub-step `Perform implantation` with regards to the communication with the ICD computational agent. Conceptually, the sub-step `Iterate attack` describes when an attacker has either gained access to an external programmer or put together a device to mimic the external programmer and then attacks the ICD by wirelessly sending all of the commands to the ICD in any order.

Figure 16 shows the step `Perform implantation` that captures the primary implantation of a new ICD into a patient within the hospital at a high level of abstraction. For the primary implantation, the major stages are represented by the sub-steps `Perform check-in to hospital`, `Perform pre-implantation tasks`, `Perform implantation tasks`, `Perform post-implantation tasks`, and `Perform check-out from hospital` and those major stages are executed sequentially. We further elaborated the sub-steps `Perform implantation tasks` and `Perform post-implantation tasks` (designated by their being step references) since they both communicate with the ICD agent while the remaining sub-steps were not further elaborated (designated by their being leaf steps). The sub-step `Perform implantation tasks` sequentially executes its sub-steps `Perform implantation procedure`, `Perform programming`, and `Perform testing`. The sub-step `Perform`

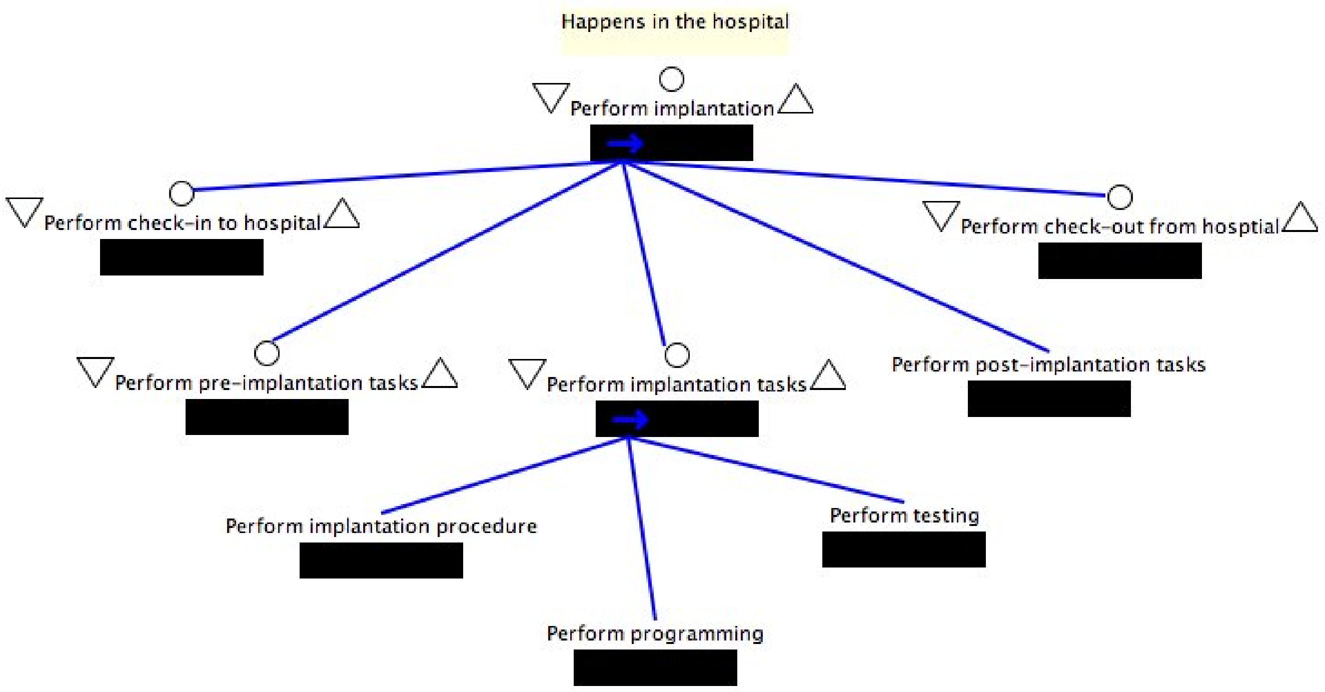
Happens in the hospital
Perform implantation
Perform check-in to hospital
Perform check-out from hosptial
Perform pre-implantation tasks
Perform implantation tasks
Perform post-implantation tasks
Perform implantation procedure
Perform testing
Perform programming

Figure 16: The step `Perform implantation` written in Little-JIL

`implantation procedure` involves physically implanting the leads and the ICD into the patient's chest cavity. After the patient's chest has been closed up, the sub-step `Perform programming` employs the wand to activate the ICD and then the external programmer to re-program the ICD's therapies as desired. For space considerations, the remainder of the ICD process coordinator model written in Little-JIL is not shown. Overall, it contains 56 steps.

ICD Property The ICD HIP requirement considers security. From [23] on page 33, the authors state the ICD HIP requirement as "an outsider should not be able to trigger an ICD's test mode, which could induce heart failure." For the ICD HIP requirement, the alphabet contains the events *EnterHCFacility*, *LeaveHCFacility*, and *TestMode*. Figure 17 shows the ICD HIP requirement as an FSA.

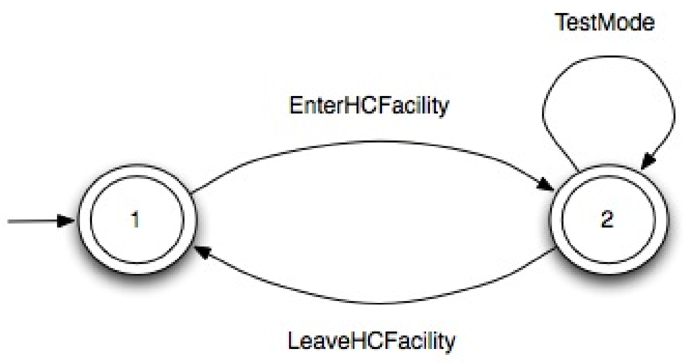

Figure 17: ICD property as an FSA

In Figure 17, the start state, state 1, represents being outside a health care facility. The transition from state 1 on event *TestMode* to the violation state represents that an outsider (i.e. attacker) should not be able to use the ICD's test mode. State 2 represents being inside a health care facility. The transition from state 1 on event *EnterHCFacility* to state 2 represents moving from outside a health care facility to inside that health care facility while the transition from state 2 on event *LeaveHCFacility* to state 1 represents moving from inside a health care facility to outside that health care facility. The transition from state 1 on event *TestMode* to the violation state represents that an outsider (i.e. attacker) should not be able to use the ICD's test mode. The transition from state 2 on event *TestMode* to state 2 represents that an insider (i.e. health care practitioner) should be able to use the ICD's test mode.

For FLAVERS/Little-JIL, the event *EnterHCFacility* is bound to the process coordinator's step `Perform implantation` entering the STARTING state or the process coordinator's step `Perform follow up` entering the STARTING state. The event *LeaveHCFacility* is bound to the process coordinator's step `Perform implantation` entering the COMPLETING state or the process coordinator's step `Perform follow up` entering the COMPLETING state. The event *TestMode* is bound to the agent's step *Administer shock for test mode* entering the COMPLETING state. Overall, the property is satisfied when the violation state is not reached.

ICD Requirement Derivation First, we investigated whether the HIP requirement considering security is useful and whether or not the requirement derivation toolset's output is useful. The requirement derivation toolset was given the ICD HIP model composed of the ICD computational agent model described in 5.1.2 and the ICD process coordinator model shown in Figure 24, the ICD HIP property shown in Figure 17, and the ICD computational agent's behavior constraint shown in Figure 15. The requirement derivation toolset outputs the the ICD HIP violates the security HIP requirement. Figure 18 shows the step view of the ICD derived requirement where the alphabet contains the events *EnterHCFacility*, *LeaveHCFacility*, *icd_activate_succ*, *icd_activate_fail*, *icd_deactivate_succ*, *icd_deactivate_fail*, *icd_readdata_succ*, *icd_readdata_fail*, *icd_testmode_succ*, *icd_testmode_fail* and there are 5 states. In the figure, there are 4 states that are shown and the violation state that is not shown.

Since the ICD HIP violated the security HIP requirement, we used the ICD derived requirement to analyze the potential failures. We partitioned the potential failures into intentional failures due to the attacker's behavior and unintentional failures due to the health care practitioners' behavior. The intentional failures are due to the attacker activating the ICD. Specifically, the derived requirement highlights two cases where the attacker activates the ICD: the first case is before the implantation (represented by the event sequence *icd_activate_succ*) and the second case is after the implantation (represented by the event sequence *EnterHCFacility, LeaveHCFacility, icd_activate_succ*). The unintentional failure is due to a health care practitioner not deactivating the ICD before the patient leaves the health care facility (represented by the event sequence *EnterHCFacility, icd_activate_succ, LeaveHCFacility*).

To prevent the unintentional failures, we define the trusted computing base of the ICD computational agent to be the external programmer, specifically the magnet or radio signal that triggers the procedure `activate`. Figure 19 shows the trusted computing base constraint represented as an FSA. The start state, state 1, represents being outside a health care facility. The transition from state 1 on event *Activate* to the violation state represents that an outsider (i.e. attacker) should not be allowed to activate the ICD. Note that this transition is now shown in Figure 19 since for brevity the violation state and its transitions are not shown. State 2 represents being inside a health care facility. The transition from state 1 on event *EnterHCFacility* to state 2 represents moving from

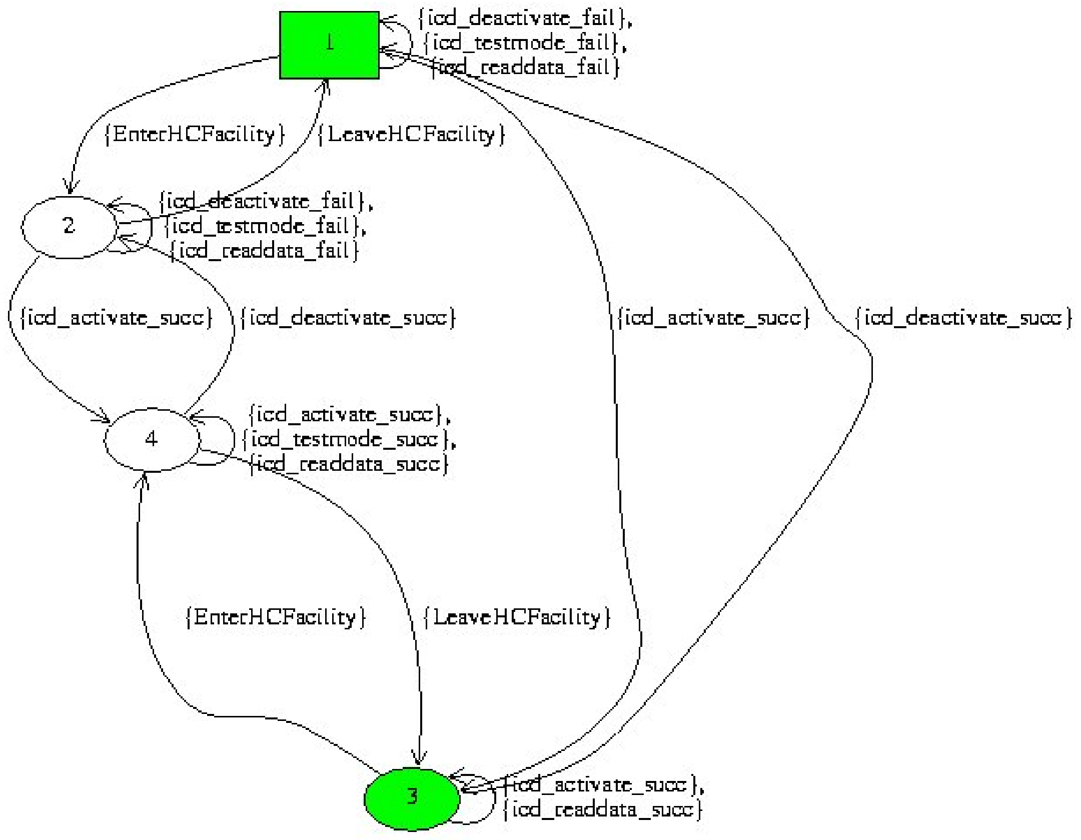

Figure 18: ICD derived requirement as an FSA

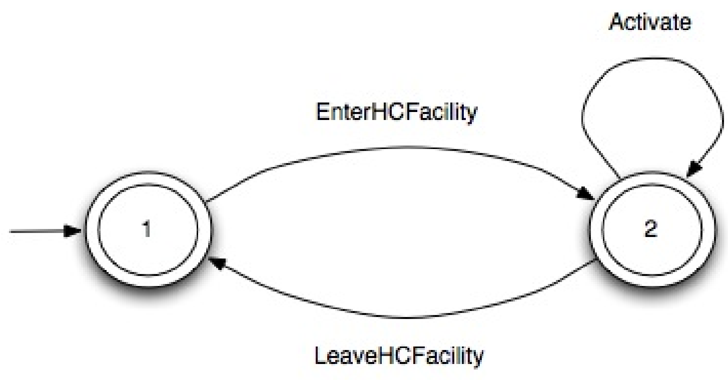

Figure 19: Trusted computing base constraint as an FSA

outside a health care facility to inside that health care facility while the transition from state 2 on event *LeaveHCFacility* to state 1 represents moving from inside a health care facility to outside that health care facility. The transition from state 2 on event *Activate* to state 2 represents that an insider (i.e. health care practitioner) should be allowed to activate the ICD.

Figure 20 shows the step view of the ICD derived requirement with the trusted computing base constraint applied. For space considerations, the violation state and its transitions are not

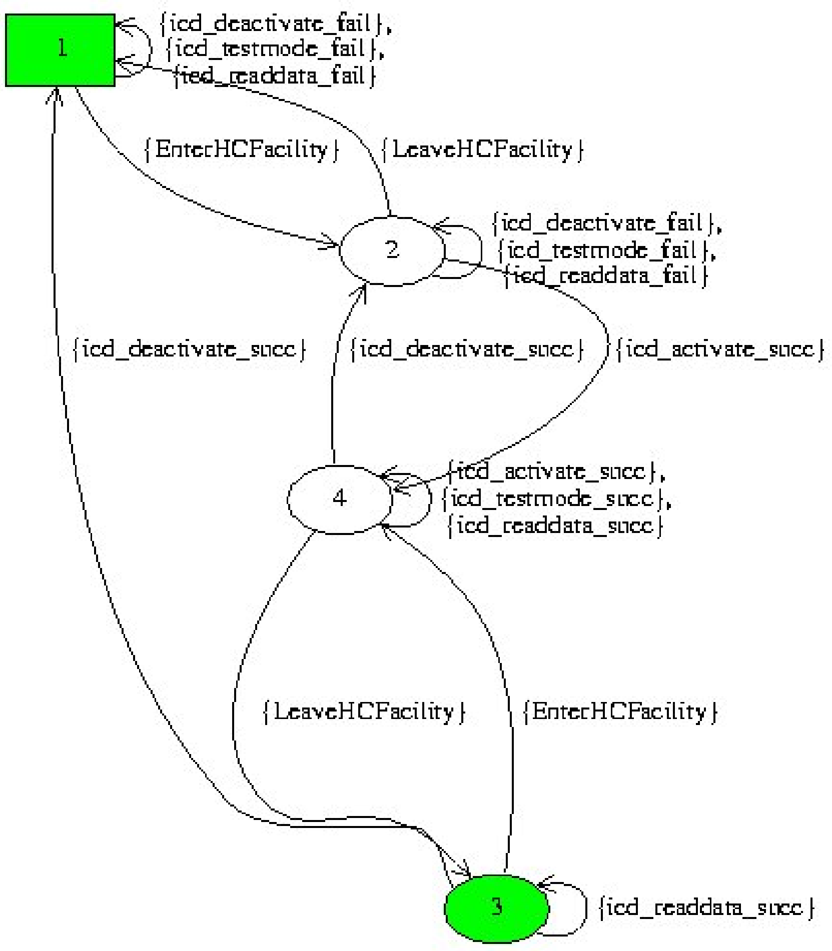

Figure 20: ICD derived requirement as an FSA when the trusted computing base constraint was applied

shown. By taking into account the trusted computing base, the derived requirement now captures that the intentional failures are prevented designated by the transition from the state 1 on event *icd_activate_succ* to the violation state (not explicitly shown) and also the transition from state 3 on event *icd_activate_succ* to the violation state (not explicitly shown). But the unintentional failure designated by the event sequence *EnterHCFacility, icd_activate_succ, LeaveHCFacility, icd_testmode_succ* is not prevented.

To prevent both intentional and unintentional failures as suggested in [23], the ICD developers could be provided with an ICD requirement that states that the ICD must deactivate itself after a certain condition is met (e.g. if the patient sits up then deactivate, if the patient leaves the operating/exam room then deactivate). In practice, we are told that such an ICD requirement is implemented, specifically the condition is that after a short period of time elapses (i.e under several minutes) then the ICD deactivates itself. It was implemented not to improve security but to reduce power consumption but regardless both the intentional and unintentional failures would be prevented.

To investigate whether the requirement derivation approach can be applied to "real" HIPs, we compared the two ICD HIPs described above with an ICD HIP that was further elaborated by adding the ICD computational agent command `writedata`. The ICD HIP with the command `writedata` was created by copying the ICD HIP. Next, it was changed so that the command `writedata` was defined and used. The ICD agent defines the procedure `writedata` that allows the external programmer to write either telemetry or therapies to the ICD. The ICD process coordinator uses the procedure `writedata` within the hospital and clinic to program the ICD and also uses it when attacking the ICD.

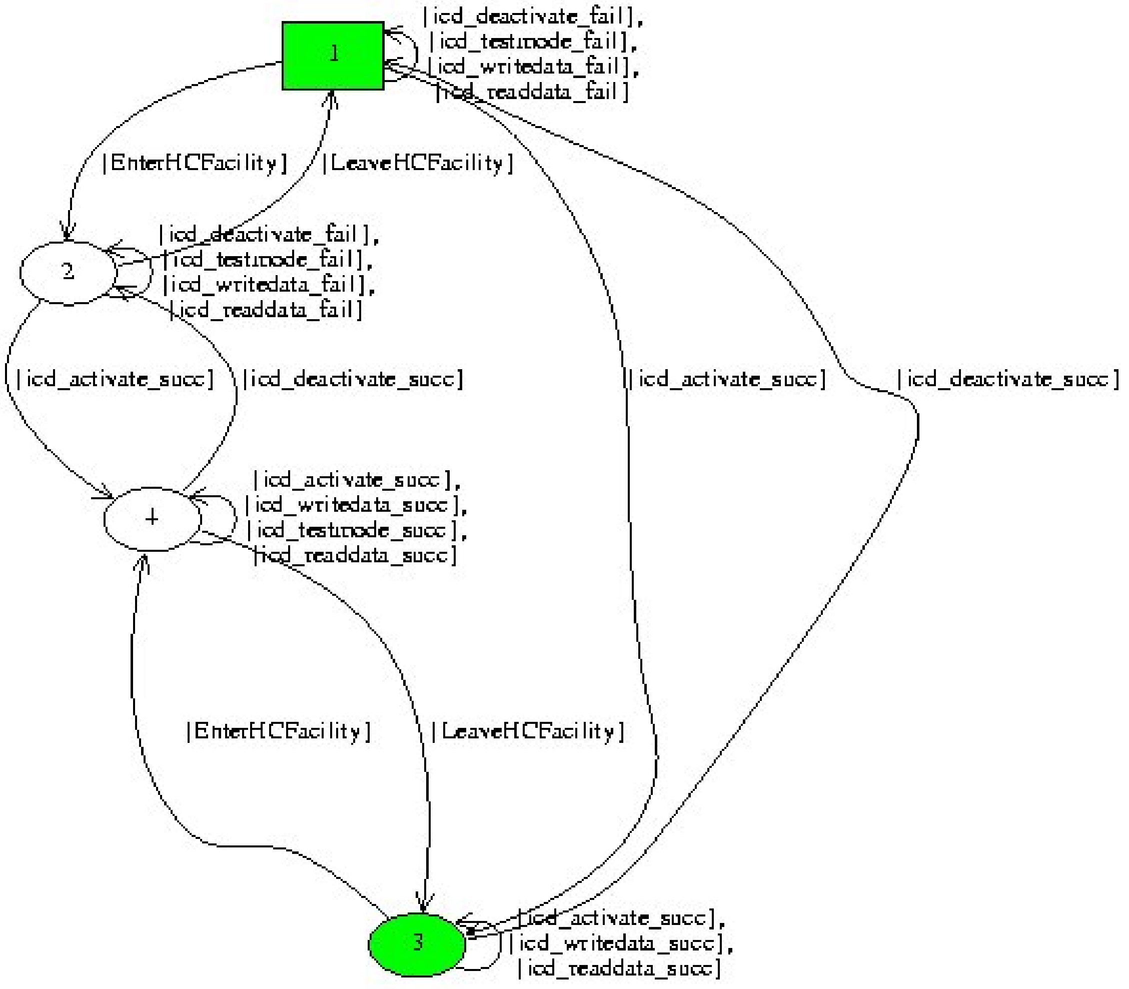

Figure 21: Derived requirement FSA for the ICD HIP with procedure `writedata`

For the ICD HIP with procedure `writedata`, Figure 21 shows the ICD computational derived requirement as an FSA, which is different from the derived requirement of the computational agent shown in Figure 18. Specifically, the alphabet contains two new events *icd_writedata_fail* and *icd_writedata_succ*, there are no new states, and there are new transitions for the two new events where the procedure `writedata` behaves like the procedure `readdata`.

Table 2 shows the the data collected for the ICD case study requirement derivation toolset runs. For the first two ICD HIPs, the derivation space needed was less than 213 MB and the derivation time needed was under 12 minutes. But for the ICD HIP with command `writedata`, the derivation space needed was 700 MB and the derivation time needed was under 23 minutes.

VARIABLE NAME	ICD	ICD w/ TCB	ICD w/ `writedata`
Requirement derivation			
Little-JIL step count	66	67	69
State count	12	13	12
Derivation space (MB)	191	166	700
Derivation time (sec)	497	594	1332
L* algorithm			
Alphabet size	8	8	9
Member. query count	1582	1816	2532
Equiv. query count	10	11	10
FLAVERS/Little-JIL			
TFG node count	148 / 419	149 / 419	178 / 521
Constraint count	7 / 13	7 / 14	7 / 15
Last node-tuple count	135383 / 608702	149160 / 610517	261100 / 2298105
Sum node-tuple count	88088823	91496963	981139329

Table 2: Data collected for the ICD case study requirement derivation runs

For the ICD case study, the ICD HIPs were further elaborated with regards to the subtasks that do communicate with the computational agent. This adversely affected the requirement derivation toolset's performance in terms of space and time. Specifically, the space needed appears to be a bottleneck. But, the derived requirements were useful and provided additional information such as how the trusted computing base is defined affects the derived requirement. In an attempt to improve the requirement derivation toolset's performance in terms of space and time, we evaluated two FSV tool optimizations as described in the next section.

5.2 How the assumption learning algorithm optimizations affect the performance of the requirement derivation in terms of space and time

From Section 3, the automated requirement derivation toolset extends an assumption learning algorithm. The assumption learning algorithm employs the L* algorithm to learn the derived computational agent requirement represented as an FSA. The L* algorithm learns the FSA by interacting with a teacher that must be able to answer membership and equivalence queries. The teacher supplies those answers by using FLAVERS/Little-JIL that employs various optimizations. For the assumption derivation algorithm, we evaluated two FSV tool optimizations.

The two FSV tool optimizations that we evaluated are an alphabet refinement optimization and a partial order optimization. Both FSV tool optimizations attempt to reduce the size of the TFG (i.e. node count and/or edge count) and also the node-tuple count. In the following, we describe the alphabet refinement optimization first and the partial order reduction optimization second.

5.2.1 Alphabet refinement optimization

For FLAVERS/Little-JIL, the alphabet refinement optimization based on [16] removes parts of the program that are not relevant to the property and constraints, specifically it removes TFG nodes and local edges. The alphabet refinement optimization inputs a TFG, a property to be verified, and a set of constraints to be applied. The relevant alphabet is defined to be the set union on the property's alphabet and each constraint's alphabet. At a high level, it outputs a copy of the input TFG where any TFG node annotated with an event not contained in the relevant alphabet is removed but all paths through that node are re-added. For each HIP process coordinator, we elaborated those steps that interacted with the agent and often did not elaborate the remaining steps. Conceptually, we manually performed alphabet refinement where the relevant alphabet consisted of the property events and the channel/parameter related events.

We evaluated on six HIPs: the pump HIP where the pump process coordinator was less/more detailed, the ICD HIP where the ICD process coordinator was less/more detailed, the ICD HIP where the ICD process coordinator was more detailed with the trusted computing base (TCB), and the ICD HIP where the ICD process coordinator was more detailed with command `writedata`. For the ICD case study, the ICD process coordinator model that was more detailed elaborated the non-leaf step `Perform implantation procedure`. The major subtasks involved are preparing the ICD before implanting it, gaining access to the chest cavity, installing the electrical leads, and closing the chest cavity. The ICD process coordinator model that was less detailed was created by changing non-leaf step `Perform implantation procedure` into a leaf step. For each HIP, the requirement derivation toolset was run as described in Section 5.

For the pump/ICD HIP with less/more detail, the derived requirements were the same. For the ICD HIP with the trusted computing base, the derived requirement captured that the trusted computing base prevented the intentional failures. For the ICD HIP with command `writedata`, the derived requirement was the same as for the ICD HIP, except there were transitions added for the command `writedata` that behaved similarly to the command `readdata`.

HIP name	Step count	Node count	Constraint count	Node-tuple sum	Time (sec)
Pump w/ less	34	174 / 233	9	6106459	245
Pump w/ more	75	174 / 248	9	6208842	249
ICD w/ less	54	148 / 419	13	88088823	490
ICD	66	148 / 419	13	88088823	497
ICD w/ TCB	67	149 / 419	14	91496963	594
ICD w/ `writedata`	69	178 / 521	15	981139329	1332

Table 3: Data collected for the alphabet refinement optimization

Table 3 shows the data collected where the first column contains the HIP name, the second column contains the detail level (either more or less), the third column contains the number of Little-JIL steps, the fourth column contains the number of TFG nodes (module 1 / module 2), the fifth column contains the memory usage as the node-tuple sum, and the sixth column contains the run time in seconds (sec). For the pump case study, the alphabet refinement optimization recognizes that the human agents communicating with the computational agent is relevant to the HIP requirement but that the human agents' other sub-tasks are not relevant so can be refined away. But for the ICD case study, the elaboration involves additional communications with the ICD computational agent that are relevant to the HIP requirement so those communications cannot be refined away. Also, the elaboration needed additional constraints to be applied that adversely affected the requirement derivation's performance in terms of space and time. To preliminarily investigate scalability, we evaluated a partial order reduction optimization described in the next section.

5.2.2 Partial order reduction optimization

The partial order reduction optimization adapted from [30] reduces the number of thread-interleavings that must be considered to verify whether or not the program satisfies the property. Within the TFG, the extra edges that capture the thread inter-leavings are called MIP (may immediately precede) edges. A MIP edge captures that execution of an event in one thread may immediately precede execution of an event in another thread. Thus, the partial order reduction optimization actually removes MIP edges. For FLAVERS/Little-JIL, the MHP (may happen in parallel) algorithm [29] uses data flow analysis to conservatively compute an approximation of the TFG node pairs that may happen in parallel with eachother. We compare a basic MIP edge strategy that adds MIP edges between all such pairs and a partial order MIP edge strategy that only adds MIP edges between particular pairs as described in [30].

For the requirement derivation, we employed both MIP strategies to compare/contrast their performance in terms of space and time. We evaluated on four HIPs: the pump HIP that was more detailed, the ICD HIP, the ICD HIP with the trusted computing base (TCB), and the ICD HIP with command `writedata` . For each HIP, the automated requirement derivation was run twice as described in Section 5. The first run was with the basic MIP strategy and the second run was with the partial order reduction (POR) MIP strategy.

HIP name	MIP strategy	MIP edge count	Memory usage (MB)	Derivation time (sec)
Pump	basic	1474 / 3218	40	249
Pump	POR	900 / 1885	32	130
ICD	basic	1175 / 4765	191	497
ICD	POR	755 / 2510	113	260
ICD w/ TCB	basic	1269 / 4765	166	594
ICD w/ TCB	POR	816 / 2510	109	311
ICD w/ `writedata`	basic	1501 / 5794	700	1332
ICD w/ `writedata`	POR	914 / 3249	526	659

Table 4: Data collected for the partial order reduction optimization

For each HIP, the derived computational agent requirement was the same for both runs. Table 4 shows the data collected where the first column contains the HIP name, the second column contains the MIP strategy (either basic or POR), the third column contains the MIP edge count, the fourth column contains the memory usage in megabytes (MB), and the fifth column contains the derivation time in seconds (sec). Overall, the partial order reduction optimization improved the performance of the requirement derivation. It decreased both the space and time used for the requirement derivation. For the HIP models, the channels are shared among the human agents and the computational agent while the parameters are local to either the human agents or the computational agent. The partial order reduction optimization is benefitting from the parameters being local. To attempt to improve the performance further, the requirement derivation approach could employ additional FLAVERS/Little-JIL optimizations.

5.3 Discussion

From our experience putting together the HIP models, it would be helpful if Little-JIL supported additional channel operations such as a synchronous send/receive and a selective receive (i.e. the process writer provides the selective receive with a query that the received data must match). For FLAVERS/Little-JIL, it may be beneficial to represent the channels at a higher-level of abstraction to attempt to improve the performance of the requirement derivation. For the security HIP requirements, the case studies may suggest a common pattern that allows certain procedures to be called within a particular setting (i.e. in a health care facility) but disallows those procedures to be called outside that setting (i.e. attacker).

From our observations, the derived computational requirements are understandable and appear to be useful to software engineers and security/privacy analysts. Initial requirements on the behavior of the computational agent were necessary for the derived computational requirements to be useful. The HIP requirement can consider safety and security. For the ICD case study, a privacy HIP requirement that states that an outsider should not be able to read data from the ICD would be quite similar to the security HIP requirement that we evaluated. The requirement derivation approach can be applied to HIPs that are defined at a high-level of abstraction. For the requirement derivation toolset runs, the derivation space needed was under 526 MB and the derivation time needed was under 23 minutes. The requirement derivation toolset that combines the L* algorithm with FLAVERS/Little-JIL benefitted from the FLAVERS/Little-JIL optimizations. For the performance, it appears that space is more of an issue than time as demonstrated by the ICD HIP with the procedure `writedata`. For a better understanding of the automated requirement derivation approach, a more extensive evaluation is needed.

6 Related Work

As mentioned in Section 2, the assumption learning algorithms, e.g. [3, 9, 14], employ the L* algorithm in combination with an FSV tool to learn the assumptions that are represented as automata. They differ with regards to the system FSV model, the requirement specification language, the FSV tool employed, the optimizations applied, and the automata (e.g, regular, tree) that are output. In contrast, the requirement derivation approach inputs a HIP composed of a process coordinator and a computational agent and also a HIP requirement that may consider safety, security, or privacy. It extends the assumption learning algorithm from [14] so the learned assumption is represented as an FSA. The extension basically weakens the learned assumption in an effort to make the learned assumption a useful computational agent requirement.

The interface synthesis algorithms, e.g. [4,8], input a software library written as a Java class and a safety requirement. Such algorithms synthesize an interface of that software library represented as an FSA where an interface is defined as "the most general way of invoking the methods in the class so that the safety property is not violated" [4]. They also employ the L* algorithm in combination with an FSV tool to synthesize the interfaces. But the interface synthesis algorithms differ with regards to the FSV tool employed. The researchers involved in [3] developed, with another researcher, the interface synthesis algorithm in [4]. For both projects, the researchers used an off-the-shelf FSV tool. On the other hand, the interface synthesis algorithm based on learning in [8] used a new FSV tool that was implemented specifically to support the learning algorithm. Additionally, Beyer et al [8] consider two other interface synthesis algorithms, one based on game theory and one based on counterexample guided abstraction refinement. For the requirement derivation approach, the computational agent is being treated like the software library and additionally the process coordinator is being taken into account. The requirement may consider safety but additionally security or privacy. The requirement derivation approach is applied to the computational agent design and not to a released computational agent. The requirement derivation approach extension is very similar to how the interface synthesis algorithms employ the L* algorithm but differs in how the teacher's equivalence query is implemented and the FSV tool employed.

The black box checking approaches, e.g. [17, 22, 32], test a given software component implementation against a specified requirement. Specifically, the software component is treated as a black box that consumes the inputs and produces the outputs but otherwise its implementation is unknown. The test results are used to build a software component model represented as an FSA. Such approaches employ the L* algorithm in combination with a testing tool to build the software component model. For [22], the black box checking approach may be provided with an initial software component model and the test results are used to update that software component model. For [17, 22], the requirement considers safety. For [32], the requirement considers security. For the requirement derivation approach, a computational agent implementation is not provided but initial requirements on its behavior may be. The requirement may consider safety, security, or privacy. Another difference is that the L* teacher employs an FSV tool instead of a testing tool.

The interactive requirement learning techniques, e.g. [2, 25], input a set of initial requirements represented as conditional actions and then iteratively refine those requirements. On each iteration, such techniques interact with the user to guide the refinement. Specifically, the user is provided with suggested refinements and must select from among those suggestions. The suggested refinements are computed by employing a learning algorithm (not L*) in combination with an AI search algorithm. For [2], the AI search algorithm is provided by an FSV tool. Similarly, the requirement derivation approach inputs any initial requirements of the computational agent's behavior and iteratively refines them. But in contrast, the requirement derivation approach only interacts with the user on input and final output while the interactive learning techniques additionally interact with the user on each iteration.

For a given software system, a security/privacy policy basically captures what the authorized flow of information through that system could be. Similar to the FSV approaches, the security/privacy information flow analysis techniques, e.g. [26, 27], dynamically or statically check whether a given software system adheres to a user-defined security/privacy policy. This cleanly separates the software system's application code from the security/privacy policy checking code. The check is performed by using an information flow analysis [7, 15]. Overall, the information flow analysis techniques primarily focus on the data flow while the requirement derivation approach primarily focuses on the control flow. But, the requirement derivation approach inputs a HIP and a security/privacy HIP requirement and outputs a derived requirement that is essentially a security/privacy policy. So in theory, the HIP and the derived requirement could then be provided to the information flow analysis techniques to be checked.

Because the security/privacy policies are challenging to reason about, there has been work on automated analysis of such policies, e.g. [19, 34]. In this work, one such automated policy analysis inputs a set of user-defined security/privacy policies and support queries over those policies. Another such automated policy analysis is a change-impact analysis that differences two versions of the same policy. The security/privacy policy extraction technique described in [35] inputs a software system written in C and employs a static analysis to extract a security/privacy policy that captures the conditional information flow. Informally, conditional means that the control flow dependencies are taken into consideration along with the data flow dependencies. Additionally, a change-impact analysis algorithm is described. At a high-level, the requirement derivation approach has a similar goal where it inputs a HIP and employs a static analysis to derive a computational agent requirement. But, the requirement derivation approach takes the HIP into account and additionally a HIP requirement into account.

7 Conclusions and Future Work

In this work, we investigated an automated requirement derivation approach. Specifically, the automated requirement derivation approach inputs a HIP composed of a process coordinator and a computational agent, a HIP requirement, and any initial requirements of the computational agent's behavior. It essentially uses a static analysis to output whether the HIP satisfies the HIP requirement and additionally a computational agent requirement represented as an FSA. The

HIP requirements may consider safety, *security, or privacy.* The HIP model is composed of a coordinator model *and an explicit computational agent model.* Additionally, the coordinator model may *explicitly define an attacker.* In addition, the requirement derivation approach can be applied during the design phase as opposed to the testing or released phase. This could help reduce development costs.

The requirement derivation toolset implements the automated requirement derivation approach except for the HIP model building. To preliminarily evaluate the approach, the toolset was applied to two case studies. From the preliminary evaluation, the automated requirement derivation approach shows promise. The derived computational agent requirements are understandable and appear to be useful to software engineers and security/privacy analysts. But, initial requirements on the behavior of the computational agent were necessary for the derived computational agent requirements to be useful. It appears that the approach can handle (at least some) safety and security properties. It was applied to "real" HIPs that are defined at a high-level of abstraction. For the requirement derivation toolset runs, the derivation space needed was under 526 MB and the derivation time needed was under 23 minutes. The requirement derivation toolset that combines the L* algorithm with FLAVERS/Little-JIL improved its performance in terms of space and time by employing two of the FLAVERS/Little-JIL optimizations.

The proposed automated requirement derivation approach could be implemented for other process modeling languages, requirement specification languages, assumption learning algorithms, and FSV tools. As future work, the process model builder tool should be implemented as described in Section 4.3.1. The requirement deriver tool should attempt to improve the performance in terms of space and time by employing additional FLAVERS/Little-JIL optimizations. To further assist the computational agent developers, the derived requirement FSAs could be used to generate positive scenarios (i.e. sequences from the start state to an accepting state) and negative scenarios (i.e. sequences from the start state to a non-accepting state).

We may investigate making the assumption learning algorithm interactive so that on each iteration the user of the toolset is provided with the current assumption and is then allowed to provide incremental requirements of the computational agent that further restrict the computational agent's behavior as suggested by [2, 25]. The automated requirement derivation approach is based on a learning algorithm. Alternatively, it could be based on a game theoretic algorithm or a counterexample guided abstraction refinement algorithm as was done in [8].

Additionally, the automated requirement derivation approach should be further evaluated. We only considered two HIPs in the medical domain. It would be interesting to consider HIPs from other domains such as the governance domain (i.e. elections). The evaluation should investigate HIPs that are more elaborated and in general the requirement derivation's scalability. Additional HIP requirements should be gathered, especially privacy HIP requirements.

From a software engineering perspective, possible future directions are to investigate HIP models where the resource repository is explicitly modeled, HIP requirements that consider resources (especially the agents), and information flow analysis. From a security/privacy perspective, possible future directions are to investigate additional static analyses that take the HIP model into account and perhaps to identify security/privacy patterns that commonly occur within the security/privacy HIP requirements.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Dalal Alrajeh, Jeff Kramer, Alessandra Russo, and Sebastin Uchitel. Learning operational requirements from goal models. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 265–275, Washington, DC, USA, 2009. IEEE Computer Society.

[3] Rajeev Alur, P. Madhusudan, and Wonhong Nam. Symbolic compositional verification by learning assumptions. In Etessami and Rajamani [18], pages 548–562.

[4] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for Java classes. *SIGPLAN Not.*, 40(1):98–109, 2005.

[5] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[6] George S. Avrunin, Lori A. Clarke, Elizabeth A. Henneman, and Leon J. Osterweil. Complex medical processes as context for embedded systems. *SIGBED Rev.*, 3(4):9–14, 2006.

[7] D.E. Bell and L.J. LaPadula. Secure computer systems, Technical Report M74-244, MITRE Corporation, Bedford, Massachusetts, USA. May 1973.

[8] Dirk Beyer, Thomas Henzinger, and Vasu Singh. Algorithms for interface synthesis. In *CAV*, Lecture Notes in Computer Science, pages 4–19. Springer, 2007.

[9] Sagar Chaki, Edmund Clarke, Nishant Sinha, and Prasanna Thati. Automated assume-guarantee reasoning for simulation conformance. In Etessami and Rajamani [18], pages 534–547.

[10] Bin Chen, George S. Avrunin, Elizabeth A. Henneman, Lori A. Clarke, Leon J. Osterweil, and Philip L. Henneman. Analyzing medical processes. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 623–632, New York, NY, USA, 2008. ACM.

[11] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security and Privacy*, 2:76–79, 2004.

[12] Stefan Christov, George S. Avrunin, Lori A. Clarke, Philip L. Henneman, Jenna L. Marquard, and Leon J. Osterweil. Using event streams to validate process definitions, Department of Computer Science, University of Massachusetts, Amherst, MA 01003, January 2009.

[13] Jamieson M. Cobleigh, George S. Avrunin, and Lori A. Clarke. Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. *ACM Trans. Softw. Eng. Methodol.*, 17(2):1–52, 2008.

[14] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In *TACAS '03: Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346, New York, NY, USA, 2003. Springer-Verlag Berlin Heidelberg.

[15] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[16] Matthew B. Dwyer, Lori A. Clarke, Jamieson M. Cobleigh, and Gleb Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 13(4):359–430, 2004.

[17] Edith Elkind, Blaise Genest, Doron Peled, and Hongyang Qu. Grey-box checking. In Elie Najm, Jean-Franois Pradat-Peyre, and Vronique Donzeau-Gouge, editors, *FORTE*, volume 4229 of *Lecture Notes in Computer Science*, pages 420–435. Springer, 2006.

[18] Kousha Etessami and Sriram K. Rajamani, editors. *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*. Springer, 2005.

[19] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 196–205, New York, NY, USA, 2005. ACM.

[20] Kevin Fu. Research notes about implantable medical devices, 2006.

[21] Graphviz: graph visualization software. http://www.graphviz.org/.

[22] Alex Groce, Doron Peled, and Mihalis Yannakakis A. Adaptive model checking. In *TACAS '02: Proceedings of the Eighth International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 357–370. Springer-Verlag Berlin Heidelberg, 2002.

[23] Daniel Halperin, Thomas S. Heydt-Benjamin, Kevin Fu, Tadayoshi Kohno, and William H. Maisel. Security and privacy for implantable medical devices. *IEEE Pervasive Computing*, 7(1):30–39, 2008.

[24] Daniel Halperin, Thomas S. Heydt-Benjamin, Benjamin Ransford, Shane S. Clark, Benessa Defend, Will Morgan, Kevin Fu, Tadayoshi Kohno, and William H. Maisel. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 129–142, Washington, DC, USA, 2008. IEEE Computer Society.

[25] Patrick Gage Kelley, Paul Hankes Drielsma, Norman Sadeh, and Lorrie Faith Cranor. User-controllable learning of security and privacy policies. In *AISec '08: Proceedings of the 1st ACM workshop on Workshop on AISec*, pages 11–18, New York, NY, USA, 2008. ACM.

[26] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans, Kaashoek Eddie, and Kohler Robert Morris. Information flow control for standard os abstractions. In *In SOSP*, 2007.

[27] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[28] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. Data flow analysis for checking properties of concurrent java programs. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 399–410, New York, NY, USA, 1999. ACM.

[29] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. An efficient algorithm for computing mhp information for concurrent java programs. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 338–354, London, UK, 1999. Springer-Verlag.

[30] Gleb Naumovich, Lori A. Clarke, and Jamieson M. Cobleigh. Using partial order techniques to improve performance of data flow analysis based verification. In *IN PROCEEDINGS OF THE ACM SIGPLAN-SIGSOFT WORKSHOP ON PROGRAM ANALYSIS FOR SOFTWARE TOOLS AND ENGINEERING*, pages 57–65, 1999.

[31] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *STOC '89: Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 411–420, New York, NY, USA, 1989. ACM.

[32] Guoqiang Shu and David Lee. Testing security properties of protocol implementations - a machine learning based approach. In *ICDCS '07: Proceedings of the 27th International Conference on Distributed Computing Systems*, page 25, Washington, DC, USA, 2007. IEEE Computer Society.

[33] Borislava I. Simidchieva, Matthew S. Marzilli, Lori A. Clarke, and Leon J. Osterweil. Specifying and verifying requirements for election processes. In *dg.o '08: Proceedings of the 2008 international conference on Digital government research*, pages 63–72. Digital Government Society of North America, 2008.

[34] Scott D. Stoller, Ping Yang, C R. Ramakrishnan, and Mikhail I. Gofman. Efficient policy analysis for administrative role based access control. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 445–455, New York, NY, USA, 2007. ACM.

[35] Michael Carl Tschantz and Jeannette M. Wing. Extracting conditional confidentiality policies. In *SEFM '08: Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 107–116, Washington, DC, USA, 2008. IEEE Computer Society.

[36] Alexander Wise. Little-JIL 1.5 language report, Department of Computer Science, University of Massachusetts, Amherst, MA 01003, 2006.

8 Appendix

8.1 Requirement Derivation Toolset

8.1.1 Derived requirement abstraction tool algorithm

Within the case study derived requirements, the other pattern matches non-leaf steps while the RPC pattern matches a pair of leaf steps. Additionally for the case study HIPs, the step sequencing used in either sequential or try. So by the Little-JIL semantics, we can apply the derived requirement abstraction since the other events will not be interleaved between the RPC related events. In more detail, if the other pattern matches a:

- non-leaf step that is an ancestor of a pair of RPC leaf steps, then by the step hierarchy that non-leaf step's execution states happen strictly before or after its predecessors execution states
- non-leaf step that is not an ancestor of a pair of RPC leaf steps, then by the step sequencing being either sequential or try that non-leaf step's execution states happen strictly before or after the RPC leaf step's execution states

The derived requirement R is represented as a deterministic FSA that is total. Let R = $(\Sigma, S, \delta, s_0, A, s_{trap})$ where:

- Σ is the alphabet that consists of the set of events of interest
- S is the set of states
- $\delta : S$ x $\Sigma \rightarrow S$ is the transition function that is deterministic and total
- $s_0 \in S$ is the unique start state
- $A \subseteq S$ is the set of accepting states
- $s_{trap} \in S$ is the unique trap state that is a non-accepting state and a sink state

Additionally, the user provides the RPCDecoding, which is a map from an RPC related event pair to a RPC related step event. Let Σ_{steps} be the set of RPC related step events. Let RPCDecoding be represented as a function from Σ x Σ to Σ_{steps}.

The AbstractDerivedRequirement(R,RPCDecoding) algorithm pseudo code is:

```
0 $states : S \rightarrow S'$
// Initialize $R' = (\Sigma', S', \delta', s'_0, A', s'_{trap})$
1 $R' \leftarrow CreateFSA(R, states)$
// Create the new alphabet, the set of other events, and the set of RPC related events
2 $\Sigma_{other} \leftarrow CreateAlphabet(A, A')$
3 $\Sigma_{RPC} \leftarrow \emptyset \cup \Sigma$
4 $\Sigma_{RPC} \leftarrow \Sigma_{RPC} \setminus \Sigma_{other}$
// Create the remaining states and transitions
5 foreach ($s \in S$) do
// Match the other pattern
6 foreach ($e \in \Sigma_{other}$) do
7 $t \leftarrow \delta(s, e)$
8 $s' \leftarrow CreateState(s)$; $t' \leftarrow CreateState(t)$
9 $\delta'(s', e) \leftarrow t'$
endforeach
// Match the RPC pattern
10 foreach ($e_1 \in \Sigma_{RPC}$) do
11 $t \leftarrow \delta(s, e_1)$
12 foreach ($e_2 \in \Sigma_{RPC}$) do
13 $u \leftarrow \delta(t, e_2)$
14 $e \leftarrow RPCDecoding(e_1, e_2)$
15 if ($e \neq null$) then
16 $s' \leftarrow CreateState(s)$; $u' \leftarrow CreateState(u)$
```

```
            17 $\delta'(s', e) \leftarrow u'$ endif
        endforeach
    endforeach
endforeach
18 DeleteDead($R'$); DeleteUnreachable($R'$) // As described by Aho, Sethi, Ullman in [1]
19 MakeTotal(R,R');
20 return $R'$
```

The CreateFSA(R,states) function pseudo code is:

```
1 $\Sigma' \leftarrow \emptyset$ // Create a new empty alphabet
2 $S' \leftarrow \emptyset$ // Create a new empty set of states
3 $s'_0 \leftarrow CreateState(states, S', s_0, A, A')$ // Copy the start state
4 $A' \leftarrow \emptyset$ // Create a new empty set of accepting states
5 $s'_{trap} \leftarrow CreateState(states, S', s_{trap}, A, A')$ // Copy the trap state
6 return $R'$
```

The CreateState(states,S',s,A,A') function pseudo code is:

```
1 $s' \leftarrow states(s)$ // Check if the given state has already been copied
2 if ($s' = null$) then
    // Allocate a new state
    3 $s' \leftarrow new$
    4 $S' \leftarrow S' \cup \{s'\}$
    5 $states(s) \leftarrow s'$
    6 if ($s \in A$) then
        7 $A' \leftarrow A' \cup \{s'\}$
    endif
endif
8 return s'
```

The CreateAlphabet($\Sigma, \Sigma', RPCDecoding$) function pseudo code is:

```
1 $\Sigma_{other} \leftarrow \emptyset \cup \Sigma$ // The set of other events
2 foreach ($e_1 \in \Sigma$) do
    3 foreach ($e_2 \in \Sigma$) do
        4 $e \leftarrow RPCDecoding(e_1, e_2)$
        5 if ($e \neq null$) then
            6 $\Sigma' \leftarrow \Sigma' \cup \{e\}$
            7 $\Sigma_{other} \leftarrow \Sigma_{other} \setminus \{e_1, e_2\}$
        endif
    endforeach
endforeach
13 $\Sigma' \leftarrow \Sigma' \cup \Sigma_{other}$
14 return $\Sigma_{other}$
```

The MakeTotal(R,R') function pseudo code is:

```
// Create the trap state and transitions
1 $s'_{trap} \leftarrow new; S' \leftarrow S' \cup \{s'_{trap}\};$
2 foreach ($s' \in S'$) do 3 foreach ($e' \in \Sigma'$) do
        4 $t' \leftarrow \delta'(s', e')$
```

```
5 if ($t' = null$) then
6 $\delta'(s', e') \leftarrow s'_{trap}$
endif
endforeach
endforeach
```

8.2 Evaluation

8.2.1 Case studies

Pump example Figure 22 shows the step "Perform in-patient surgery" written in Little-JIL.

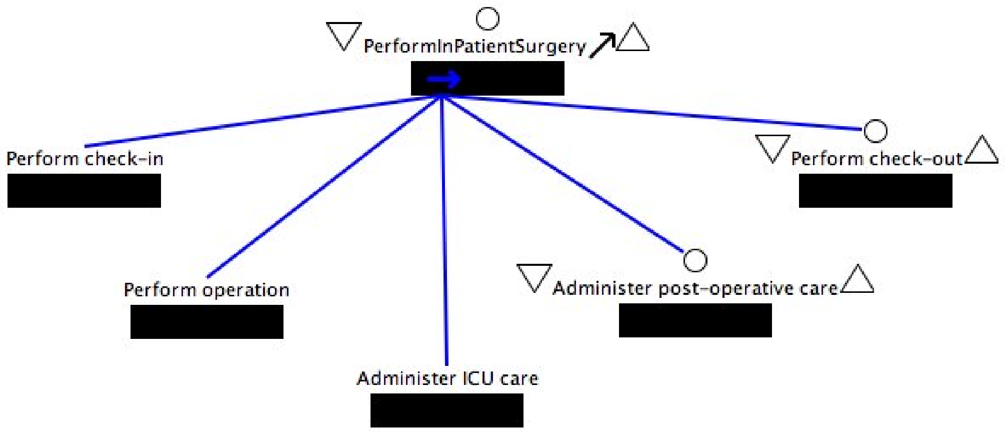

Figure 22: Step `Perform in-patient surgery` written in Little-JIL

ICD that was more detailed example Figure 23 shows the pessimistic ICD computational agent model written in Little-JIL.

Figure 24 shows the ICD process coordinator model written in Little-JIL where the step `PerformICDPatientCare` is the entry point (designated by the northeast arrow to the right of the step name).

Figure 25 shows step `Iterate attacker` briefly described in Section 5.1.2.

Figure 26 shows step `Perform implantation procedure` described in Section 5.1.2.

After physically implanting the ICD, Figure 27 shows the step `Perform testing` that ensures that the ICD performs its monitoring and therapies. The step `Perform testing` sequentially executes its sub-steps `Review telemetry and therapies` and `Test therapies`. The step `Review telemetry and therapies` executes sequentially so it first executes sub-step `Perform read data` that reads the telemetry and therapies from the ICD and second executes sub-step `Confirm read data` that confirms that the telemetry and therapies are appropriate before proceeding to testing the therapies. The step `Test therapies` also executes sequentially so it first executes sub-step `Perform test mode` and then sub-step `Confirm therapies applied`. For the nominal control flow, the test mode administers a shock to the heart that stops the heart (i.e. enters cardiac ventriculation), the ICD's therapies are triggered to restart the heart, and the test completed successfully For the exceptional control flow, the sub-step `Perform test mode` throws a TestFailedException when the ICD does not administer the shock to the heart and the sub-step `Confirm therapies applied`

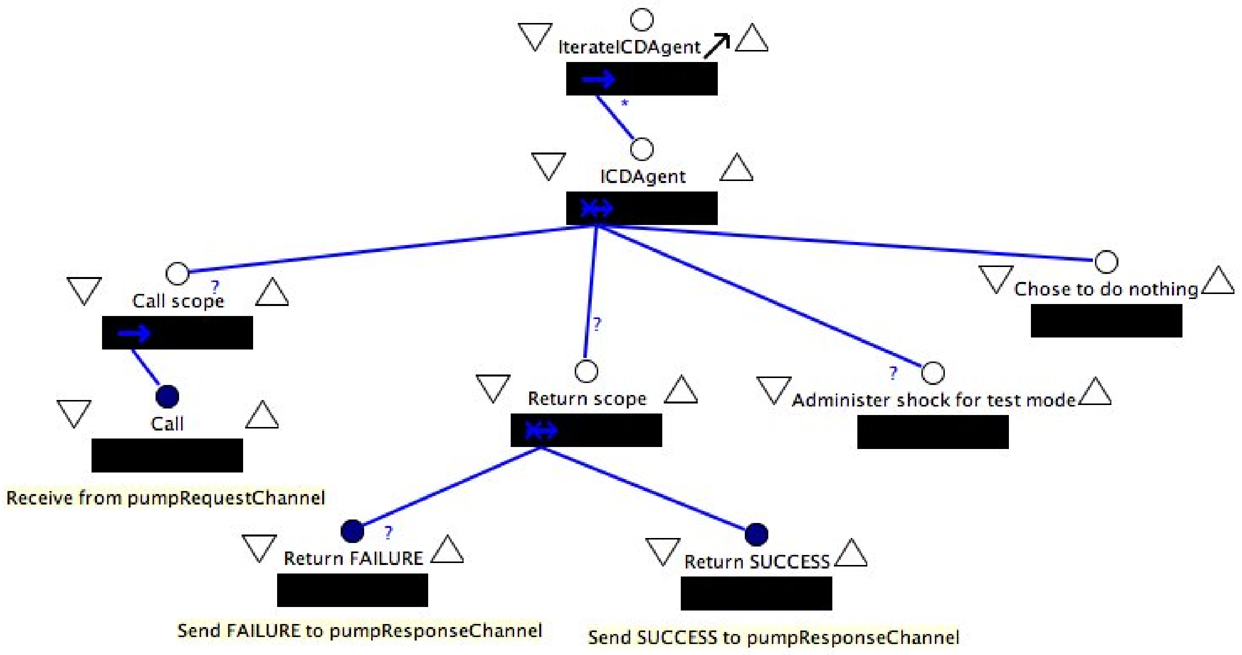
IterateICDAgent
*
ICDAgent
?
Call scope
Call
Receive from pumpRequestChannel
?
Return scope
?
Return FAILURE
Send FAILURE to pumpResponseChannel
Return SUCCESS
Send SUCCESS to pumpResponseChannel
?
Administer shock for test mode
Chose to do nothing

Figure 23: Pessimistic ICD computational agent model written in Little-JIL

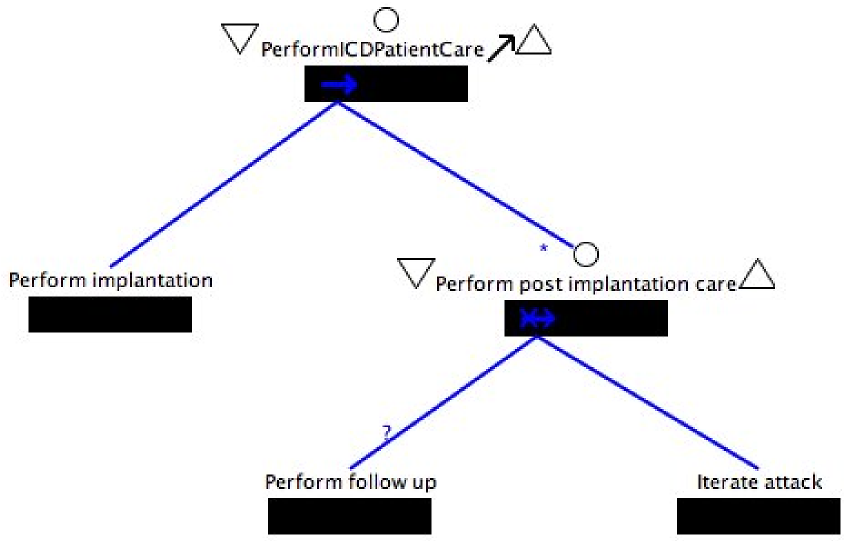

Figure 24: ICD process coordinator model written in Little-JIL

throws a TestFailedException when the ICD's therapies are not triggered and therefore the heart is not restarted. The TestFailedException is propagated to the step "Test therapies" where the exception handler `Handle test failed` catches the TestFailedException and afterwards completes the step `Test therapies`.

For the ICD computational agent derived requirement, the step abstraction tool inputs the following:

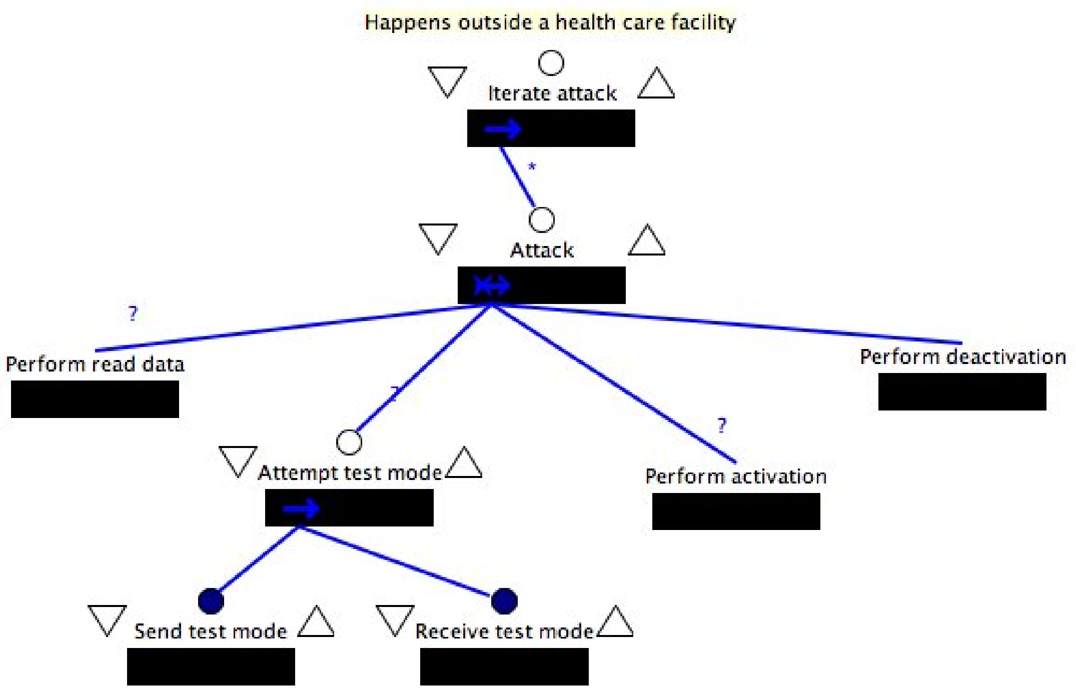

Figure 25: Step `Iterate attacker` written in Little-JIL

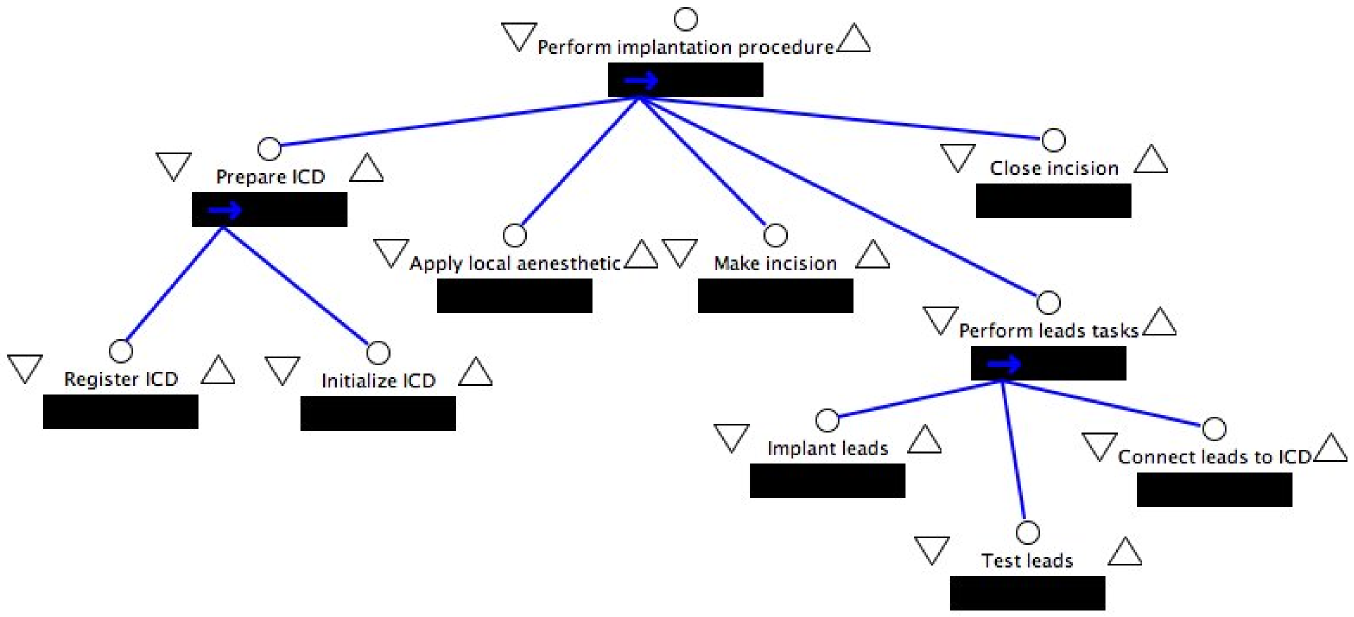

Figure 26: Step `Perform implantation procedure` written in Little-JIL

- event *icdRequestChannel=0* paired with event *icdResponseChannel==0* maps to event *icd_activate_succ*
- event *icdRequestChannel=0* paired with event *icdResponseChannel==1* maps to event *icd_activate_fail*

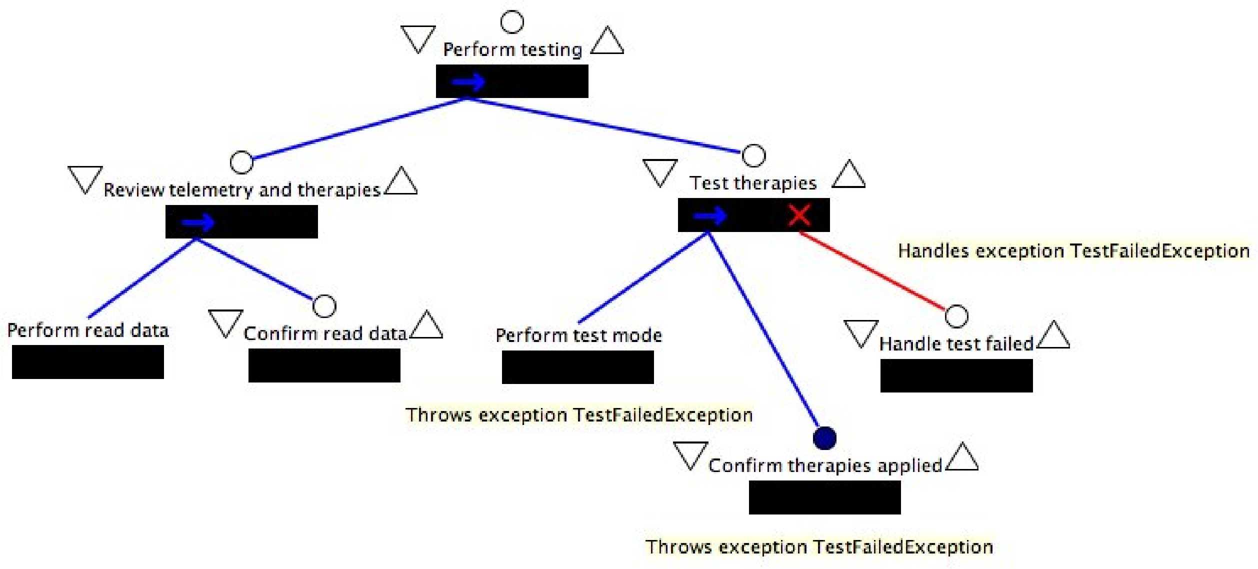
Perform testing
Review telemetry and therapies
Test therapies
Handles exception TestFailedException
Perform read data
Confirm read data
Perform test mode
Handle test failed
Throws exception TestFailedException
Confirm therapies applied
Throws exception TestFailedException

Figure 27: Step `Perform testing` written in Little-JIL

- event *icdRequestChannel=1* paired with event *icdResponseChannel==0* maps to event *icd_deactivate_succ*
- event *icdRequestChannel=1* paired with event *icdResponseChannel==1* maps to event *icd_deactivate_fail*
- event *icdRequestChannel=2* paired with event *icdResponseChannel==0* maps to event *icd_readdata_succ*
- event *icdRequestChannel=2* paired with event *icdResponseChannel==1* maps to event *icd_readdata_fail*
- event *icdRequestChannel=3* paired with event *icdResponseChannel==0* maps to event *icd_testmode_succ*
- event *icdRequestChannel=3* paired with event *icdResponseChannel==1* maps to event *icd_testmode_fail*

ICD that was more detailed with the trusted computing base example For the ICD computational agent derived requirement, the step abstraction tool takes the inputs as described in the previous paragraph.

ICD that was more detailed with procedure `writedata` example For the ICD computational agent derived requirement, the step abstraction tool inputs the following:

- event *icdRequestChannel=0* paired with event *icdResponseChannel==0* maps to event *icd_activate_succ*
- event *icdRequestChannel=0* paired with event *icdResponseChannel==1* maps to event *icd_activate_fail*
- event *icdRequestChannel=1* paired with event *icdResponseChannel==0* maps to event *icd_deactivate_succ*
- event *icdRequestChannel=1* paired with event *icdResponseChannel==1* maps to event *icd_deactivate_fail*
- event *icdRequestChannel=2* paired with event *icdResponseChannel==0* maps to event *icd_readdata_succ*
- event *icdRequestChannel=2* paired with event *icdResponseChannel==1* maps to event *icd_readdata_fail*
- event *icdRequestChannel=3* paired with event *icdResponseChannel==0* maps to event *icd_testmode_succ*

- event *icdRequestChannel=3* paired with event *icdResponseChannel==1* maps to event *icd_testmode_fail*
- event *icdRequestChannel=4* paired with event *icdResponseChannel==0* maps to event *icd_writedata_succ*
- event *icdRequestChannel=4* paired with event *icdResponseChannel==1* maps to event *icd_writedata_fail*